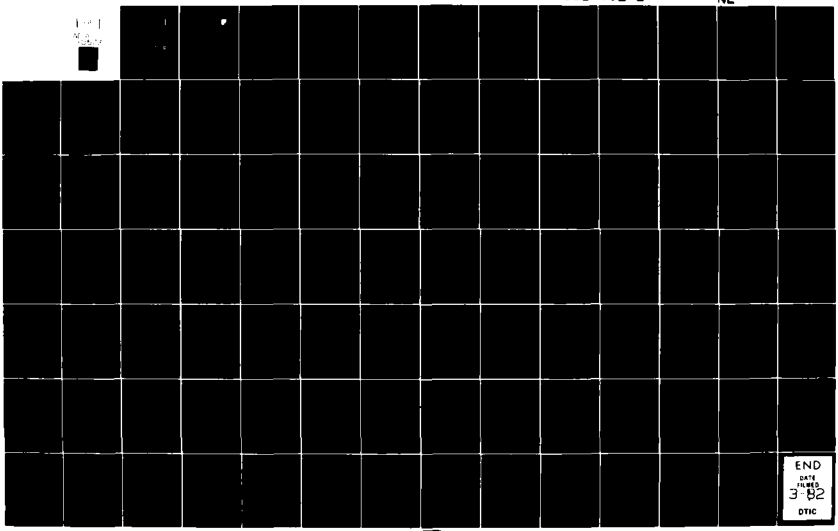
AD-A110 573 INTERMETRICS INC CAMBRIDGE MA F/G 9/2
ADA INTEGRATED ENVIRONMENT I COMPUTER PROGRAM DEVELOPMENT SPECI--ETC(U)
DEC 81 F30602-80-C-0291
UNCLASSIFIED RADC-TR-81-358-VOL-2 NL



PHOTOGRAPH THIS SHEET

AD A110573

DTIC ACCESSION NUMBER

LEVEL

INVENTORY

Intermetrics, Inc
Cambridge, MA
ADA Integrated Environment I Computer Program Development Specification. Interim Rept. 15 Sep. 80 - 15 Mar. 81
RADC-TR-81-358 Vol. II Dec 81
Contract F30602-80-C-0291

DOCUMENT IDENTIFICATION

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

DISTRIBUTION STATEMENT

ACCESSION FOR	
NTIS GRA&I	☒
DTIC TAB	☐
UNANNOUNCED	☐
JUSTIFICATION	
BY	
DISTRIBUTION /	
AVAILABILITY CODES	
DIST	AVAIL AND/OR SPECIAL
A	

DISTRIBUTION STAMP

DTIC
S ELECTE D
JAN 25 1982
D

DATE ACCESSIONED

82 01 12 018

DATE RECEIVED IN DTIC

PHOTOGRAPH THIS SHEET AND RETURN TO DTIC-DDA-2

DTIC FORM OCT 79 70A

DOCUMENT PROCESSING SHEET

RADC-TR-81-358, Vol II (of seven)
Interim Report
December 1981

ADA INTEGRATED ENVIRONMENT I COMPUTER PROGRAM DEVELOPMENT SPECIFICATION

Intermetrics, Inc.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: RADC-TR-81-358, Vol II (of seven) — 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): ADA INTEGRATED ENVIRONMENT I COMPUTER PROGRAM DEVELOPMENT SPECIFICATION	5. TYPE OF REPORT & PERIOD COVERED: Interim Report 15 Sep 80 - 15 Mar 81
	6. PERFORMING ORG. REPORT NUMBER: N/A
7. AUTHOR(s)	8. CONTRACT OR GRANT NUMBER(s): F30602-80-C-0291
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Intermetrics, Inc. 733 Concord Avenue Cambridge MA 02138	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 62204F/33126F 55811908
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (COES) Griffiss AFB NY 13441	12. REPORT DATE: December 1981
	13. NUMBER OF PAGES: 87
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Donald F. Roberts (COES)

Subcontractor is Massachusetts Computer Assoc.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Ada	MAPSE	AIE
Compiler	Kernel	Integrated environment
Database	Debugger	Editor
KAPSE	APSE	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The Ada Integrated Environment (AIE) consists of a set of software tools intended to support design, development and maintenance of embedded computer software. A significant portion of an AIE includes software systems and tools residing and executing on a host computer (or set of computers). This set is known as an Ada Programming Support Environment (APSE). This B-5 Specification describes, in detail, the design for a minimal APSE, called a MAPSE. The MAPSE is the foundation upon which an

DD FORM 1473, 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

APSE is built and will provide comprehensive support throughout the design, development and maintenance of Ada software. The MAPSE tools described in this specification include an Ada compiler, linker/loader, debugger, editor, and configuration management tools. The kernel (KAPSE) will provide the interfaces (user, host, tool), database support, and facilities for executing Ada programs (runtime support system).

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

This document was produced under Contract F30602-80-C-0291 for the Rome Air Development Center. Mr. Don Roberts is the COTR for the Air Force. Dr. Fred H. Martin is Project Manager for Intermetrics.

RADC-TR-81-358, Volume II (of seven) has been reviewed and is approved for publication.

APPROVED: Donald F. Roberts

DONALD F. ROBERTS
Project Engineer

APPROVED:

JOHN J. MARCINIAK, Colonel, USAF
Chief, Command and Control Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

Table of Contents

1. SCOPE 1
1.1 Identification 1
1.2 Functional Summary 1

2. APPLICABLE DOCUMENTS 1
2.1 Government Documents 1
2.2 Non-Government Documents 2

3. REQUIREMENTS 3
3.1 Program Definition 3
3.2 Detailed Functional Requirements 3

3.2.1 Database Elements 4
3.2.1.1 Objects 4
3.2.1.2 Configurations and Partitions 6
3.2.1.3 Window Objects 6
3.2.1.4 Special Kinds of Composite Objects 7
3.2.1.5 System-defined Attribute Category 8
3.2.1.6 System-defined Attribute Access Control 9
3.2.1.7 System-defined Attribute History 11

3.2.2 Storage Representation of Objects 12

3.2.3 Operations on Content of Database Objects 13
3.2.3.1 Operations on Simple Objects 13
3.2.3.2 Operations on Composite Objects 17
3.2.3.3 Operations on Window Objects 20
3.2.3.4 Copying/Renaming Operations 22

3.2.4 Operations on Attributes 23
3.2.4.1 Operations on User-defined Attributes 23
3.2.4.2 Category Operations 24
3.2.4.3 Access Control 28
3.2.4.4 History/Archiving Operations 29

3.2.5 Other Database Operations 32
3.2.5.1 Synchronization 32
3.2.5.2 Configurations Reporting and Management 34
3.2.5.3 Backup and Recovery 36

3.2.6 Program Invocation and Control 37
3.2.6.1 Program Context 37
3.2.6.2 Parameter Passing 39
3.2.6.3 Private Object Operations 40
3.2.6.4 Interprogram Communication 42
3.2.6.5 Debugging and Control Interface 43
3.2.6.6 Exception Handling 44

INTERMETRICS INCORPORATED • 733 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

3.2.7 KAPSE User Interface 44
3.2.7.1 Overall User View of the Database 44
3.2.7.2 LOGIN System 45
3.2.7.3 User Accounting 48
3.2.7.4 The Inter-User Mail System 49
3.2.7.5 User Terminal Handling 50

3.2.8 Ada Run Time System and High-Level I/O 52
3.2.8.1 Ada Tasking Model 52
3.2.8.2 Storage Management 53
3.2.8.3 Package INPUT_OUTPUT 53
3.2.8.4 Package TEXT_IO, Interactive I/O Extensions 53
3.2.8.5 Package FORMATTED_IO 55

3.2.9 KAPSE/Host Interface -- VM/370 and OS/32 56
3.2.9.1 Overall Architecture 56
3.2.9.2 Physical Disk I/O 57
3.2.9.3 Terminal I/O 59
3.2.9.4 Device Input/Output and Import/Export 61
3.2.9.5 Ada Tasking Support 62
3.2.9.6 Program Loading and Initiation 63
3.2.9.7 Interprogram Communication 64

3.3 Adaptation and Rehosting 65
3.3.1 Installation Parameters 65
3.3.2 Operation Parameters 65
3.3.3 System Capacities 65
3.3.4 Rehosting Requirements 65

4. QUALITY ASSURANCE 66
4.1 Ada Machine Testing 67
4.2 Production Input/Output Tests 67
4.3 KAPSE Version 1 Test Case Generation 67
4.4 K1 Reliability Test 68
4.5 Full Function Testing 68
4.6 KAPSE Version 3 Testing 68
4.7 Acceptance Testing 68

10. APPENDIX 69
10.1 Package INPUT_OUTPUT in Ada 69
10.2 Ada Tasking Model in Ada 75

INTERMETRICS INCORPORATED • 733 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

1.0 SCOPE

1.1 Identification

This specification establishes the requirements for performance, design, test, and qualification of a set of computer program modules identified as the Kernel Ada Programming Support Environment (KAPSE).

1.2 Functional Summary

The KAPSE provides several facilities to the Ada Programming Support Environment (APSE), which can be grouped into the following three areas:

1. Database Operations.
2. Invocation of and communication between Ada programs.
3. Run-time support for the execution of Ada programs, including high-level input/output packages.

This specification identifies the functional capabilities of the various KAPSE modules and describes the KAPSE/tool interface as well as the KAPSE/Host computer interface.

2.0 APPLICABLE DOCUMENTS

Please note that the bracketed number preceding the document identification is used for reference purposes within the text of this document.

2.1 Government Documents

[G-1] Reference Manual for the Ada Programming Language, proposed standard document, U.S. Department of Defense, July 1980.

[G-2] Requirements for Ada Programming Support Environment, "STONEMAN," Department of Defense, February 1980.

[G-3] Statement of Work for Ada Integrated Environment, PR No. B-O-3233, December 1979.

2.2 Non-Government Documents

[N-1] IBM Virtual Machine Facility/370: System Programmer Guide, International Business Machines, Inc.

[N-2] OS/32 Programmer Reference Manual, Perkin-Elmer Computer Systems Division, Oceanport, NJ, April 1979.

[N-3] The Art of Computer Programming, V. 3, Donald Knuth, Addison Wesley, 1973.

[I-1] System Specification for Ada Integrated Environment, Type A, Intermetrics, Inc., IR-676, March 1981.

Computer Program Development Specifications, Type B5, for Ada Integrated Environment:

[I-2] Ada Compiler Phases, IR-677.

[I-3] MAPSE Command Processor, IR-679.

[I-4] MAPSE Generation and Support, IR-680.

[I-5] Program Integration Facilities, IR-681.

[I-6] MAPSE Debugging Facilities, IR-682.

[I-7] MAPSE Text Editor, IR-683.

[I-8] AIE Technical Report (Interim), IR-684.

3. REQUIREMENTS

3.1 Program Definition

The KAPSE provides database, program invocation, and run-time support for all MAPSE tools and user Ada programs. In so far as possible, the KAPSE isolates the rest of an APSE from host machine idiosyncracies, making the entire MAPSE toolset and user-developed programs easily portable from one APSE to another.

The KAPSE database is the repository for all user data and programs, as well as the primary medium of tool to tool communication and coordination. The KAPSE database facilities provide for the construction, organization, and partitioning of large configurations of inter-related program, data, and documentation elements. It records the nature and purpose of these elements, and allows for access control and synchronization. Finally, the KAPSE database facilities provide historical information recording the derivation and relations between the objects stored within the database, as well as sufficient indices to fully reconstruct from disk or archival storage the content of old or lost source text.

3.2 Detailed Functional Requirements

This section is organized as follows:

Database Concepts
1. Database Elements
2. Storage Representation of Objects

Database Facilities
3. Operations on Content of Database Objects
4. Operations on Attributes
5. Other Database Operations

Using the KAPSE
6. Program Invocation and Control
7. KAPSE User Interface

Ada Language Support
8. Ada Run Time System and High-Level I/O

Hosting the KAPSE
9. KAPSE/Host Interface --VM/370 and OS/32

3.2.1 Database Elements

3.2.1.1 Objects

The database is a collection of objects, all of which have attributes and content. These objects fall into three broad classes, simple, composite, and window. The content of a simple object is a sequence of primitive data elements, and is used to represent the concept of an Ada external file. The content of a composite object is a set of named component objects, which may themselves be either simple, composite, or window. The database as a whole is a single large composite object, whose direct components are major divisons of the database. The content of a window object is a reference to some other part of the database, with an associated access limitation.

The attributes of an object are meta-information describing its content, purpose, access control, etc. As such they provide the primary means for building, organizing, and partitioning the database. The attributes can be grouped into three classes:

1. System-defined attributes (Category, Access Control, and History);

2. user-defined distinguishing (name) attributes;

3. user-defined non-distinguishing attributes.

System-defined attributes are discussed later in this document; user-defined attributes, which have a simpler form, are discussed below:

Each user-defined attribute is represented as a pair consisting of attribute label and attribute value. For clarity, it will be written in the unabbreviated form: label => value, to be read, label "is" value. Both the label and the value of an attribute are simple strings of characters. The label must satisfy the syntax of an Ada identifier (i.e., start with a letter, and continue with letters, numbers, or underscores).

A list of attribute label/value pairs must be enclosed by parentheses, with commas separating, as shown below:

```
(PROJECT=>SHUTTLE,FUNCTIONAL_AREA=>NAVIGATION)
```

This would specify that the attribute labeled "PROJECT" has the value "SHUTTLE" and that the attribute labeled "FUNCTIONAL_AREA" has the value "NAVIGATION."

User-defined distinguishing attributes have a special use: When a composite object is created, part of its definition specifies a list of attributes by which its components will be named (i.e. distinguished from one another). When a component is created within this composite object, values for these distinguishing attributes must be supplied. These may be used later to select this component from the composite object. The new component may not be created if an existing component has the same list of distinguishing attribute values.

For example:

```
CREATE_COMPOSITE("PROJECT_LIBRARY",
    COMPONENT_DA=>"PROJECT FUNCTIONAL_AREA MODULE");
```

Now components could be created within this new composite object "named" as follows:

1. (PROJECT=>SHUTTLE,FUNCTIONAL_AREA=>NAVIGATION,
   MODULE=>INITIALIZATION)

2. (PROJECT=>SHUTTLE,FUNCTIONAL_AREA=>CONTROL,
   MODULE=>INITIALIZATION)

3. (MODULE=>INITIALIZATION,PROJECT=>VOYAGER,
   FUNCTIONAL_AREA=>NAVIGATION)

4. (MODULE=>INTERPOLATION,FUNCTIONAL_AREA=>NAVIGATION,
   PROJECT=>VOYAGER)

Two components need differ in only one of the distinguishing attribute values to be considered distinctly named (eg., (1) and (2) above).

Positional notation may be used instead of labeled notation, based on the ordering specified when the composite object was created:

1. SHUTTLE.NAVIGATION.INITIALIZATION

2. SHUTTLE.CONTROL.INITIALIZATION

3. VOYAGER.NAVIGATION.INITIALIZATION

4. VOYAGER.NAVIGATION.INTERPOLATION

INTERMETRICS INCORPORATED • 733 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

3.2.1.2 Configurations and Partitions

Composite objects are well-formed configurations of component objects, and as such can be directly manipulated by Ada programs. New components can be created, existing components can be modified or deleted. The composite object as a whole can be copied as a unit. The structure of a composite object can be mandated by a category specification, and access control can be applied to the configuration as a whole, or to its individual parts.

Composite objects may also be partitioned by specifying values for some of the attributes of their components, as follows:

(PROJECT=>SHUTTLE) would include (1) and (2) from above.

(FUNCTIONAL_AREA=>NAVIGATION,MODULE=>INITIALIZATION) would include (1) and (3).

Positional notation may also be used to specify partitions, but the special value "*" must be supplied as a place holder:

.CONTROL. would include only (2)

VOYAGER.*.* would include (3) and (4)

Both distinguishing and non-distinguishing attributes may be used to specify partitions of a composite object. Non-distinguishing attributes are not ordered, and so only the labeled notation may be used. Here is an example of a single partition specification giving values for both kinds of attributes:

(FUNCTIONAL_AREA=>NAVIGATION,PRIORITY=>HIGH)

This partition would include (1), (3), and (4) from above only if their current value for a non-distinguishing attribute labeled "PRIORITY" were "HIGH." If a non-distinguishing attribute has never been specified for a object, the value is taken to be the null string.

3.2.1.3 Window Objects

A third kind of object in the KAPSE database is called a window object. The content of this object is simply a cross-reference to all or part of some other object in the database, along with a specification of access limitations on that object. Window objects are the means by which a user may delegate access to and/or responsibility for parts of the database to other users.

To access an object, the user provides a name relative to the partition selected by some window. When a window is created, the name of the object and the partition limitation, if any, are specified. In addition, the user may specify further limits on the allowed types of access to the object (see 3.2.3.3 for an extended example).

The access limitations are expressed as a capacity (abstract role) to which all users of the window are limited. The capacity is identified by an ASCII STRING, like "MANAGER", "READER", or "PROGRAMMER." Within the partition accessible through the window, individual objects specify with a simple table (the access control attribute), what operations may be performed by which capacities. Multiple windows may exist specifying different capacities relative to the same partition.

3.2.1.4 Special Kinds of Composite Objects

(a) Program Context Object. Each program running in the MAPSE has associated with it a single composite object called its program context object. It is through the program context that Ada programs get access to the rest of the database. The program context object is normally deleted after the program finishes execution. Components of this program context may be simple objects (temporary files), composite objects (a set of temporary objects), or windows on more permanent parts of the database. All program context objects are composite objects using a single distinguishing attribute labeled LOCAL_NAME for their components.

When an Ada program creates or opens an object in the database, it specifies the name of the object as a single ASCII STRING. If the name begins with a dot, then the rest of the name is interpreted relative to the program context object.

If the name does not begin with a dot, the KAPSE requires that there be a window in the program's context with the LOCAL_NAME of CURRENT_DATA, and interprets the name relative to that window. In effect, it is as though ".CURRENT_DATA." were inserted at the front of the name.

The linker produces executable program context objects [I-5]. When such a context is to be invoked from another program context, it is first copied into the invoker's context, then parameters and window(s) are supplied, and, finally, it is initiated (see 3.2.6.1).

(b) Private Objects. A user may create a special kind of composite object, called a private object, which can be used to implement an encapsulated abstract data object analogous to an Ada object of private type. A private object is composed of a DATA component, and a number of operation components. Each operation component is an executable program context object. The operation context objects are pre-initalized with appropriate windows on all or part of the DATA object, which allows them to perform more primitive operations on the DATA component than are accessible to the normal user. For example, the KAPSE mail system allows users to send and receive mail using private objects called mailboxes, without giving users the ability to corrupt the internal structure of the mailboxes.

Operation context objects are not copied before they are initiated (as opposed to normal executable program context objects built by the linker). Instead, the INVOKE_OPERATION procedure is used to initiate the operation, and only one such invocation may be in progress at a time. If a second program attempts to initiate the same operation context object before it completes, the second program is delayed up to a specified TIME_LIMIT (see 3.2.6.3). In this way, private objects provide both encapsulation and synchronization.

3.2.1.5 System-defined Attribute Category

An object in the database records all information describing its current state. As explained above, this information includes the following:

1. Current content;
2. current list of user-defined attribute labels and values;
3. current access control attribute;
4. current category.

In addition, information recording the derivation (how and why) of the object is maintained (the history attribute).

The category of an object specifies which parts of its current state are fixed for the lifetime of the object. In this sense, the category provides a time-and state-independent classification of the object. The category does not record the entire current state but, rather, a list of restrictions on the state. For example, the category could restrict a particular non-distinguishing attribute to have only certain values, or could require certain minimum access capacities to be defined to provide certain specific access rights.

The category of an object may itself be changed without necessarily changing other parts of the state of the object, so long as the state of the object satisfies the new constraints.

Because categories are represented by a list of constraints rather than by a specific name, constraints may be added or removed from specific objects without necessarily preventing existing programs from processing them in a meaningful way.

3.2.1.6 System-defined Attribute Access Control

Every object has an access control attribute. This attribute is represented as a simple table giving, for each capacity name, a list of primitive access rights.

Here is a hypothetical access control table:

Capacity	Access Rights
OWNER	SYSTEM_ATTRIB_MODIFY -- May adjust access control -- attribute, etc.
EDITOR	READ, WRITE
TESTER	READ, COPY
READER	READ

Every window specifies a partition and a capacity. When an object is specified by an access path going through a window, the user is limited to the access rights specified for that capacity.

Objects define primitive access rights according to their class:

Object Class	Access Rights
All	ALL, NONE, COPY, SYSTEM_ATTRIB_READ, SYSTEM_ATTRIB_MODIFY, USER_ATTRIB_READ, USER_ATTRIB_MODIFY
Simple	READ, WRITE, APPEND, READ_DELETE
Composite	LIST_COMPONENTS, CREATE_COMPONENT, DELETE_COMPONENT, SELECT_COMPONENT Component_capacity
Window	GO_THROUGH
Program_context	INITIATE, PROG_CTX_CONTROL, plus composite object access rights.
Private	INVOKE Operation_context_name, plus composite object access rights.

The identifier ALL is used to represent all access rights meaningful for the class of the object.

In the case of the SELECT_COMPONENT access right for a composite object the table entry includes the internal component-relative capacity to which the user is limited. Additional restrictions on composite objects and their components are implicit in the partition specification associated with a window. In particular, LIST_COMPONENTS, SELECT_COMPONENTS, CREATE_COMPONENTS, and DELETE_COMPONENTS all are limited to referring to components within the specified partition.

USER_ATTRIB_MODIFY is also limited if the partition associated with the window specifies values of non-distinguishing attributes: A program may not change the value of a non-distinguishing attribute of an object if by so doing it makes the object inaccessible through the window.

When a program calls a KAPSE primitive procedure or function it may be implicitly exercising one of the above access rights. The KAPSE call will fail if the capacity of the window implied in the name of the object does not include the implied access right. In particular, as the KAPSE follows the access path supplied as a STRING parameter, the window effective at each point must allow SELECT_COMPONENT to access a component of a composite object, and must allow GO_THROUGH to access objects through a window.

At both of these points, the capacity associated with the window is translated to a new capacity for use in checking access along the next part of the path. These requirements are universal and are not repeated when specifying the type of window needed to perform a particular KAPSE primitive in the rest of this document.

The following capacities are pre-defined:

Capacity	Access Rights
SYSTEM	ALL; reserved for system maintenance.
OWNER	ALL unless explicitly included in access control table. A window of this capacity is given to a program on its own program context object.
INFERIOR	ALL unless explicitly included in access control table. A window named ".CALLER_CONTEXT" of this capacity is given to programs invoked from a superior program context.
WORLD	NONE unless included in access control table. In general, all users have a WORLD window on the <u>root</u> of the entire database, as well as on the TOOLS composite object.

In the default case, where no access control is explicitly specified (or implicitly specified by the category constraints), the KAPSE provides full access to all SYSTEM, OWNER, and INFERIOR windows, and no access to others (see 3.2.3.3 and 3.2.4.3 for examples).

3.2.1.7 System-defined Attribute History

From the point of view of history, two significantly different kinds of objects exist in the database: source objects and derived objects. Source objects are those text objects produced, in general, by a human using a text editor (see [I-7] for description of the Text Editor). Derived objects, text or otherwise, are those produced as the output of other tools or user programs, with little or no direct input from the user other than parameters.

The history attribute is designed to uniquely identify a particular state of an object's content. In the case of a source object, the history attribute refers to a source archive wherein an efficient representation of multiple states (revisions or versions) of the same basic text may be stored. The history attribute consists of a unique identifier of the source archive, and an index used to select the lines of text that make up this particular state of the object.

When a source object is created, it may be associated with an existing source archive, or allocated a new source archive unique identifier. When associated with an existing source archive, the source object is assigned the next sequential state index. With a new archive, the source object is assigned state number one.

When a source object is edited or deleted, the pre-modification contents and attributes of the object are merged into the source archive under the assigned state index, and the state is indicated as being recorded in the source archive. If edited rather than deleted, the new history attribute is given the next available state index, and this new index is remembered as being a successor to the old index.

Later, the state of the non-distinguishing attributes and content may be recreated as a new object, but with the same history attribute. If this object is edited or deleted, the archive indicates that it is not necessary to re-save the old content and attributes.

The history attribute of a derived object consists of a unique identifier of a program invocation script, and an index indicating which output of the program gave this state of the object. The script records the parameters specified when the program was invoked, an array of copies of the history attributes of each object read as input, and a count of the number of objects created or modified as output. If the program produces no new output states, the script may be deleted after the program completes.

The KAPSE maintains reference counts for all history scripts and archives. If specified, the KAPSE will also maintain a reference list for particular scripts or archives, thereby allowing easy tracing of all references to a particular source or derived object. The reference lists can grow quite long, so the maintenance of the lists is at the option of the user. Periodically, source archives and scripts that have not been referenced recently may be dumped to tape through a KAPSE service. Nevertheless, the KAPSE always maintains an

index of the off-line history and may be explicitly requested to re-activate specific scripts or archives. Even recently referenced archives or scripts may be written to tape to ensure that the tape contains an internally self-consistent representation of history. However, these history elements are left on-line as well.

In addition to the data mentioned above, each history script and source archive records the date and time, as well as the USER NAME, when the program execution or source editing occurred (see 3.2.4.4).

3.2.2 Storage Representation of Objects

The content and attributes of all normal objects are stored on the disk provided by the host machine. A small number of <u>device</u> objects are created by the system manager to provide direct and import/export access to physical I/O devices or disk files of the host system.

Normal data is stored in a fixed-block format on disk, independent of the user-visible record size. Each block is assigned a unique BLOCK_ID. The BLOCK_ID is used to locate the data of the block, and the reference count for the block (stored in a separate table). The KAPSE maintains a central buffer cache so that repeated references to the same disk blocks do not each require physical I/O.

Every block can be broken into four parts: a time sequence number, a level number, BLOCK_IDs, and other data. The time sequence number, which is recorded whenever a block is written, is a never-decreasing number incremented at each system backup, used to provide incremental backup at the block level.

The level number is zero for data-only blocks (no BLOCK_IDs), and otherwise is one greater than the maximum level number of all blocks ever referenced by a BLOCK_ID. BLOCK_IDs only appear in blocks managed by the KAPSE. The other data are both user data and KAPSE-managed data.

The connections implied by the BLOCK_IDs within disk blocks may form a tree structure or, in the presence of sharing, an acyclic graph. The LEVEL number of the block indicates the "distance" to the furthest leaf block. In the case of a simple object, a B-tree structure is used [N-3], resulting in a uniform depth for all leaf blocks.

When the KAPSE copies an object, it simply stores the BLOCK_ID of the root of the object in its new location, and increments the root block's reference count by one. No additional disk blocks are allocated to hold the logical copy of the object until one of the copies is actually modified.

The root block of an object includes enough BLOCK_IDs and other data to gain access to all information representing the current state and history of the object (other than its distinguishing attributes, which are stored at a higher level). Category, access control, history reference, user-defined non-distinguishing attributes, and the data content, if sufficiently short and simple, can all be stored within the root block. However, as the content and attributes grow larger or more complicated, additional blocks will be automatically allocated to hold the overflow, with BLOCK_IDs stored in the root block to point to them. For small simple objects, the entire object can fit in a single block.

When a block is to be changed, the KAPSE checks if the block is shared, by determining whether its <u>join</u> count is greater than zero. This count is the number of superior blocks (in the path followed from the root of the entire database to this block) which have a reference count greater than one. If the block is shared, a new BLOCK_ID is allocated and the changed contents of the block are written into the new physical disk block, with an initial reference count of one. This procedure is applied recursively up until a superior block with join count of zero is reached. That block is re-written in place, and the reference count of the block it used to refer to is decremented by one. The total amount of block copying is never more than if the object were entirely copied initially.

3.2.3 Operations on Content of Database Objects

3.2.3.1 Operations on Simple Objects

(a) <u>Specification</u>. The primary user-visible interface to simple objects is the standard Ada package INPUT_OUTPUT specified in the LRM [G-1, 14.1]. Package INPUT_OUTPUT is implemented in terms of a more primitive file-handling package. For more details see Appendix 10.1.

The basic primitives available are as follows:

```
Package SIMPLE_OBJECTS is

type FILE_HANDLE(SIZE_IN_BITS:  INTEGER) is record
     FH_INDEX:  INTEGER :=  -1;
end record;

type FILE_MODE is (IN_MODE, INOUT_MODE, OUT_MODE);

procedure CREATE(FH:  in out FILE_HANDLE; NAME:  in STRING;
    MODE:  in FILE_MODE);
     -- Requires a window allowing WRITE where
     --  the object is created,
     --  and CREATE_COMPONENT on
     --  the enclosing composite object.
```

```
procedure CREATE_DEVICE_OBJ(NAME: in STRING;
    HOST_DEVICE_NAME: in STRING; ROOT_WINDOW: in STRING);
     -- This procedure is provided for a system
     --  manager to set up an association between
     --  a special database object and
     --  a host file or physical I/O device.  The
     --  HOST_DEVICE_NAME is host-dependent.
     -- Requires a window allowing CREATE_COMPONENT,
     --  as well as a SYSTEM window on the root
     --  of the database (ie., system managers
     --  only, please!).

procedure DELETE(NAME: in STRING);
     -- Requires a window giving DELETE_COMPONENT on
     --  the enclosing composite object.

procedure COPY(OLDNAME: in STRING; NEWNAME: in STRING);
     -- This procedure creates a logical copy of the
     --  specified object with identical content and
     --  non-distinguishing attributes.  The
     --  distinguishing attributes of the copy are
     --  implied by NEWNAME.
     -- COPY involves no actual disk data block copying.
     --  When either the original or copy is later
     --  modified, the KAPSE makes actual physical
     --  copies of the affected blocks.

procedure OPEN(FH: in out FILE_HANDLE; NAME: in STRING;
     MODE: in FILE_MODE);
     -- Requires a window giving READ and/or WRITE
     --  depending on FILE_MODE.

procedure CLOSE(FH: in out FILE_HANDLE);

type FILE_INFO_BLOCK is record
     SIZE: FILE_INDEX;  -- See Package INPUT_OUTPUT.
     FIRST: FILE_INDEX;
     LAST: FILE_INDEX;
     NEXT_READ: FILE_INDEX;
     NEXT_WRITE: FILE_INDEX;
end record;

procedure GET_FILE_INFO(FH: in FILE_HANDLE;
    INFO: out FILE_INFO_BLOCK);

procedure SET_FILE_INFO(FH: in FILE_HANDLE,
    INFO: in FILE_INFO_BLOCK);

procedure READ(FH: in FILE_HANDLE, ADDR: in INTEGER,
    SIZE: in INTEGER; NUM_REC: out INTEGER; MAX_REC: in INTEGER);

procedure WRITE(FH: in FILE_HANDLE, ADDR: in INTEGER,
    SIZE: in INTEGER; NUM_REC: in INTEGER);

end SIMPLE_OBJECTS;
```

The generic types IN_FILE, OUT_FILE, INOUT_FILE used in Package INPUT_OUTPUT are converted to the type FILE_HANDLE with the size in bits determined by ELEMENT_TYPE'SIZE. The KAPSE deals directly in terms of records of the specified number of bits. All FILE_INDEX values are multiplied internally by the KAPSE by the SIZE_IN_BITS to get a bit offset into the content of the simple object.

Procedures READ and WRITE work on one or more records at a time, with SIZE of each record limited to be no greater than the SIZE_IN_BITS associated with the FILE_HANDLE.

(b) Internal Representation and Algorithms. The content of a simple object is represented using fixed-size blocks. Simple objects that occupy more than one physical block are stored in a multi-way B-tree structure [N-3], allowing random access to any block of the object with a small number of disk block references. Logically contiguous blocks of an object are allocated from a free block map with as close as possible to the optimal physical separation.

Two storage organizations are supported: indexed and direct access. With the indexed storage organization, data exists only for records actually written. Any attempt to READ records that do not exist results in the DATA_ERROR exception. Each record written contains its own index, and the multi-way tree structure is used to locate blocks that contain individual records. In addition, in the indexed storage organization, only the number of bits specified in the WRITE procedure call are actually allocated for the record.

With direct-access storage organization, data is assumed to exist from the first record written to the last record written. Any records that exist but have not been written contain all zero bits. For certain Ada types, attempts to READ an all-zero record may cause a DATA_ERROR or a CONSTRAINT_ERROR exception, depending on the compiler implementation. Because all intermediate records exist, the multi-way tree structure does not record index values along with physical block numbers. The index value is implicit in the position within the multi-way tree node.

The Package INPUT_OUTPUT procedure TRUNCATE causes all records after a specified LAST record to become undefined. Using SET_FILE_INFO directly (see above) it is also possible to advance the index of the FIRST record to be defined. A simple object may be used for communication between two programs in a stream fashion by one program WRITE'ing at the end of the file, thereby automatically advancing LAST, and the other READ'ing at the beginning of the file, advancing the FIRST record index when desirable. This has the effect of "throwing away" the already-read records, thus preventing the content of the object from becoming excessively large as the communication proceeds. Opening an object in SHARED_STREAM mode provides this stream facility automatically (see 3.2.5.1).

The structure of each multi-way tree node depends on the storage organization. Indexed organization nodes have the following structure:

```
type PACKED_BIT_VEC is array(NATURAL range <>) of BOOLEAN;
pragma PACK(PACKED_BIT_VEC);

BLOCK_SIZE: constant := <implementation-dependent>;
type BLOCK is new PACKED_BIT_VEC(1..BLOCK_SIZE);
type BLOCK_ID is range 0..<implementation-dependent>;

type TIME_SEQUENCE is private;

DATA_LIM: constant := BLOCK_SIZE-INTEGER'SIZE-TIME_SEQUENCE'SIZE;

IX_NODE_LIM: constant := (DATA_LIM-INTEGER'SIZE-BLOCK_ID'SIZE)/
                   (BLOCK_ID'SIZE + FILE_INDEX'SIZE);
type INDEXED_NODE is record
     LAST_WRITE: TIME_SEQUENCE;
     LEVEL: INTEGER;  -- LEVEL 1 is lowest-level node
     NUM_PTRS: INTEGER range 0..IX_NODE_LIM;
     PTRS: array (0..IX_NODE_LIM) of BLOCK_ID;
     IXS: array(1..IX_NODE_LIM) of FILE_INDEX;
end record;
```

Indexed organization leaf blocks have the following structure:

```
type INDEXED_LEAF(NUM_ELEMENTS: INTEGER, DATA_SIZE: INTEGER)
    is record
     LAST_WRITE: TIME_SEQUENCE;
     LEVEL: INTEGER := 0;  -- Leaf block is always LEVEL 0
     IXS: array(1..NUM_ELEMENTS) of FILE_INDEX;
     DATA_PTRS: array(1..NUM_ELEMENTS) of INTEGER;
         -- Index into DATA
     DATA: PACKED_BIT_VEC(1..DATA_SIZE);
end record;
    -- NUM_ELEMENTS and DATA_SIZE limited so it fits in a BLOCK.
    -- The above is meant to be suggestive.  The actual
    --  implementation is optimized so that DATA_SIZE is not
    --  actually stored with the block, but is rather calculated
    --  from NUM_ELEMENTS.  In addition, IXS and DATA_PTRS are
    --  combined into a single array.
    -- The length of a particular stored element may vary
    --  if the ELEMENT_TYPE is a variant type, but may
    --  be calculated from DATA_PTRS(N+1) - DATA_PTRS(N),
    --  with the last element packed tightly up to the end
    --  of DATA (i.e., DATA_PTRS(NUM_ELEMENTS+1)
    --  would be DATA_SIZE+1).
```

Direct-access organization multi-way tree nodes have the following structure:

```
DA_NODE_LIM: constant := DATA_LIM / BLOCK_ID'SIZE;
type DIRECT_ACCESS_NODE is record
     LAST_WRITE: TIME_SEQUENCE;
     LEVEL: INTEGER;  -- LEVEL 1 is lowest-level node
     PTRS: array(1..DA_NODE_LIM) of BLOCK_ID;
end record;
```

Direct-access organization leaf blocks have the following structure:

```
type DIRECT_ACCESS_LEAF is record
     LAST_WRITE: TIME_SEQUENCE;
     LEVEL: INTEGER := 0;  -- Leaf block is always LEVEL 0.
     DATA: PACKED_BIT_VEC(1..DATA_LIM);
end record;
     -- Number of elements per leaf is always
     --  DATA_LIM/ELEMENT_TYPE'SIZE.
     -- Individual elements are slices of DATA.
```

Locating a particular record within an indexed organization file involves the standard multi-way search algorithm, starting with the root and selecting the appropriate branch based on the value of the desired FILE_INDEX. The search requires (log N)/(log IX_NODE_LIM) disk block references, where N is the total number of blocks in the file (for a complete discussion of multi-way searching, see [N-3]).

Locating a particular record in the direct access organization first requires converting the desired FILE_INDEX into a bit offset. This is then divided by BLOCK_SIZE, giving a block number. This is adjusted according to the current number of levels in the tree and the FIRST defined logical record FILE_INDEX. The search requires (log N)/(log DA_NODE_LIM) disk block references, where N is the total number of blocks in the file.

3.2.3.2 Operations on Composite Objects

(a) Specification. The following primitives are available for creating and modifying composite objects:

```
Package COMPOSITE_OBJECTS is

procedure CREATE_COMPOSITE(NAME: in STRING;
    COMPONENT_DA: in STRING);
     -- COMPONENT_DA is a space separated list of attribute
     --  labels required of all components created in the object.
     -- Requires a window giving CREATE_COMPONENT on the enclosing
     --  composite object.
```

```
procedure DELETE(NAME: in STRING);
     -- Requires a window giving DELETE_COMPONENT on the enclosing
     --  composite object.

type PARTITION_HANDLE is private; -- Similar to FILE_HANDLE.

procedure OPEN_PARTITION(PH: in out PARTITION_HANDLE;
    NAME: in STRING);
     -- NAME is a specification of a partition,
     --  like "(PROJECT=>SHUTTLE)" or "*.CONTROL.*"
     -- Requires a window giving LIST_COMPONENT on the composite
     --  object implied by the partition.

procedure CLOSE_PARTITION(PH: in out PARTITION_HANDLE);

procedure GET_PARTITION_INFO(PH: in PARTITION_HANDLE;
    INFO: out PARTITION_INFO_BLOCK);
     -- Returns miscellaneous INFO about the partition,
     --  including the number of components currently in
     --  the partition, the FIRST, LAST, and NEXT component
     --  names (in ASCII lexicographic order), etc.

function GET_NEXT_COMPONENT(PH: in PARTITION_HANDLE) return STRING;
     -- This returns the name of the next component of the given
     --  partition, as a parenthesized list of distinguishing
     --  attribute values in a single STRING.  The names
     --  are returned in ASCII lexicographic order.
```

Operations that create and delete components of a composite object implicitly modify its content. The name of the object specified to CREATE and CREATE_COMPOSITE determines the composite object in which it is created.

Database names passed as parameters to procedures such as CREATE, OPEN, and DELETE are used to locate the object within the database. As explained above (see 3.2.1.4), names that start with a dot are interpreted relative to the program's context object, and those that do not are interpreted as though ".CURRENT_DATA." were inserted at the front of the name. If the desired object is a component of a composite object, first the name of the composite object is given, followed by a dot, followed by the distinguishing attributes of the component.

The distinguishing attributes may be specified using positional notation, with dots separating, or with label=>value notation, inside parentheses and with commas separating. If the object is a component of a component, and the first and second set of distinguishing attribute labels are distinct, then the label=>value notations for the two sets may be merged into one. For example, the following could all be equivalent:

```
(PROJECT=>SHUTTLE,AREA=>NAVIGATION).(UNIT=>A,SUBUNIT=>B)
(PROJECT=>SHUTTLE,AREA=>NAVIGATION,UNIT=>A,SUBUNIT=>B)
(UNIT=>A,AREA=>NAVIGATION,SUBUNIT=>B,PROJECT=>SHUTTLE)
(AREA=>NAVIGATION,PROJECT=>SHUTTLE).A.B
SHUTTLE.NAVIGATION.(SUBUNIT=>B,UNIT=>A)
SHUTTLE.NAVIGATION.A.B
```

After specifying the access path to the object, the STRING passed to OPEN or CREATE may be used to convey extra information. The additional information is in the form of a parenthesized label=>value list separated from the access path by a space character. With this syntax, it is possible to specify the following extra information:

```
Call            Extra labeled information
----            -------------------------
CREATE          RESERVE_MODE, ACCESS_CONTROL, CATEGORY_SPEC
OPEN            RESERVE_MODE
```

For example:

```
OPEN ( FILE1, "STREAM_OBJECT_1  (RESERVE_MODE=>SHARED_STREAM)" );
CREATE ( FILE2, "PUBLIC_INFO_FILE" &
    "  (ACCESS_CONTROL=>(WORLD=>(READ,APPEND)))" );
```

(b) Internal Representation and Algorithms. The content of a composite object is represented as a multi-attribute tree of component objects. The distinguishing attributes are handled in the order specified when the composite object was created. Each attribute introduces an additional level into the tree, where each level itself has a B-tree indexed organization, with variable length keys corresponding to the distinguishing attribute values.

Because each level in the multi-attribute tree has an indexed organization, the KAPSE provides fast (log N) access to components of even large composite objects.

(c) Examples.

```
CREATE_COMPOSITE("COMP", "MODULE  RELEASE_NUM");

CREATE(FH, "COMP.(MODULE=>DISPLAY, RELEASE_NUM=>1)", OUT_MODE);
CLOSE(FH);

OPEN(FH, "COMP.DISPLAY.1", IN_MODE); -- Using positional notation.
CLOSE(FH);

OPEN_PARTITION(PH, "COMP.*.1"); -- Scan through partition.
STR := GET_NEXT_COMPONENT(PH);
PUT_LINE("First component of COMP is: " & STR);
-- On the user's terminal should appear:
-- "First component of COMP is (MODULE=>DISPLAY,RELEASE_NUM=>1)"
CLOSE_PARTITION(PH);
```

3.2.3.3 Operations on Window Objects

(a) Specification. A window object is created by specifying its name, the target object to be referenced, a common ancestor node label, an optional partition specification relative to the target object, and an optional capacity name which indicates an additional access limitation.

```
procedure CREATE_WINDOW(NAME: in STRING; TARGET: in STRING;
   COMMON_ANCESTOR: in STRING := ""; PARTITION: in STRING := "";
     CAPACITY: in STRING := "");

procedure DELETE(NAME: in STRING);

procedure COPY(OLDNAME: in STRING; NEWNAME: in STRING);

procedure REVOKE(SUPER_WINDOW: in STRING; SUB_WINDOW: in STRING);
     -- This incapacitates the specified SUB_WINDOW if
     --  it is was derived from the specified SUPER_WINDOW.
```

(b) Internal Representation and Algorithms. A window is a cross-reference to another object to be "seen" through the window. This cross-reference is recorded as an access path from the window to the target object, going through a common ancestor composite object (node) in the database composite object hierarchy tree.

After a composite object is created, it may be given a hierarchy node label. The access path from a window specifies the hierarchy node label of the appropriate common ancestor node, and then the path back down the hierarchy to the target object. This is analogous to the use of block and package identifiers in Ada to specify the path to a selected component [G-1, 4.1.3 (c) and (e)]. The hierarchy node label is a non-distinguishing attribute of the composite object, called NODE_LABEL.

Copies of a window may only be stored at points in the hierarchy where the hierarchy node label refers to the same ancestor composite object as the original window. In any case they may only be stored at points below the named common ancestor node. By selecting a particular common ancestor node, the creator of a window effectively limits the dispersion of copies of the window. If the common ancestor chosen is the root of the entire hierarchy tree, copies may be stored anywhere in the database.

To allow easy tracing of windows, a window cross-reference table is maintained at every node which is used as a common ancestor. Whenever a window is copied, a new entry is added to the window cross-reference table giving the location of the new window and the object seen. To trace all windows giving some kind of access to an object, the KAPSE scans only through the window cross-reference tables of the ancestor nodes of the object (as opposed to a complete search of the database hierarchy tree).

When a new window is created (not simply a copy), the access rights of the new window are necessarily a subset of the access rights of some pre-existing parent window implicit in the path to the target object specified in the CREATE_WINDOW call. The new window is added to the designated common ancestor node's window cross-reference table and is indicated as a sub-window of the implicitly identified parent window. The newly designated common ancestor must be the same as, or a descendant of, the ancestor designated by the parent window. If the COMMON_ANCESTOR parameter to CREATE_WINDOW is the null string, the parent window's common ancestor is assumed. At a later time, a program may REVOKE the access granted by the creator of the sub-window by using the parent window (or a copy of it). If all copies of a parent window are deleted, its parent window inherits the revocation right over the sub-windows.

If a composite object has any components that are windows, it lists the hierarchy node labels of all higher-level nodes used as common ancestors by those windows. When such a composite object is copied, its copy must remain a descendant of all of the listed ancestor nodes. A composite object may not be given a NODE_LABEL attribute value which would change the interpretation of any descendant window's reference path.

If a window, its target, and their designated common ancestor are all components of the composite object, no special processing is done on COPY, and the new copy of the window component refers to the new copy of the target object.

If the common ancestor is not a component of the copied composite object then the new copy of the window component continues to refer to the old target object. In this case, the window cross-reference table at the common ancestor node is updated to indicate the presence of an additional window.

Windows do not store any kind of internal identifier of their target objects or common ancestor nodes, but rather the ASCII string access path. This implementation ensures that copying windows and composite objects containing windows does not require changing internal identifiers. Furthermore, it allows logical copies of any object to share the same disk data blocks. Only when a part of a logical copy is changed are new disk blocks allocated to hold the changed data.

To ensure that a sub-window does not exceed the rights of its parent, the content of a window records not only its current capacity, but also the set of capacities of its progenitors. When a window is used, it is limited to access rights legal for its own capacity and all of its progenitors. The KAPSE calculates this intersection of rights by performing a simple bit-wise AND of the appropriate capacity access-right bit maps.

(c) Examples.

```
CREATE_WINDOW(".WORKSPACE", "SHUTTLE.NAVIGATION.INIT");
     -- This creates a convenient shorthand window
     --  named .WORKSPACE.

OPEN(FILE, ".WORKSPACE.SPEC");
     -- This is equivalent to:
     --  OPEN(FILE, "SHUTTLE.NAVIGATION.INIT.SPEC");

CREATE_WINDOW(".RESTRICTED_WORKSPACE", ".WORKSPACE.",
    CAPACITY=>"WORLD");
     -- Create sub-window limited to access rights
     --  given to the WORLD.

OPEN(FILE2, ".RESTRICTED_WORKSPACE.SPEC");
     -- This may fail if SHUTTLE.NAVIGATION.INIT.SPEC
     --  doesn't give READ or WRITE access to the WORLD.

CREATE_WINDOW(".SMALLER_VIEW", ".WORKSPACE.",
    PARTITION=>"(TEST_LEVEL=>2)");
     -- The window .SMALLER_VIEW only lets its user
     --  see objects with attribute TEST_LEVEL having
     --  a value of 2.
```

3.2.3.4 Copying/Renaming Operations

(a) Specification. The following operations are defined for copying and renaming objects:

```
procedure COPY(OLDNAME: in STRING; NEWNAME: in STRING);
     -- This procedure is used to make a copy of an existing
     --  object (simple, composite, or window).  If the object
     --  is a window, or contains a window with an external
     --  common ancestor, the new copy may only be created
     --  where it is still a descendant of the common ancestor(s).
     -- The new copy shares disk blocks with the original object
     --  until one of them is changed.
     -- Requires a window giving COPY on OLDNAME, and
     --  CREATE_COMPONENT at NEWNAME.
     -- It will also fail if NEWNAME already exists,
     --  or if OLDNAME contains any running program
     --  contexts.

procedure DELETE(NAME: in STRING);
     -- This procedure is used to delete any object
     --  (simple, composite, window).
     -- Requires a window giving DELETE_COMPONENT on
     --  the enclosing composite object.
     -- Running program contexts are also aborted
     --  and deleted by this primitive.
```

```
procedure RENAME(OLDNAME: in STRING; NEWNAME: in STRING);
     -- A call on this procedure is exactly identical to
     --  COPY(OLDNAME, NEWNAME); DELETE(OLDNAME);
     -- with the same restrictions.
```

(b) Internal Representation and Algorithms. The design of the KAPSE database facilitates copying by providing block reference counting at a low level (see 3.2.2). Copying an object first involves checking that it is legal, and then simply incrementing the reference count of the root block of the object and adding its new name and BLOCK_ID to the implied composite object. Additional blocks are allocated only when the original or copy is later modified, and then only enough blocks to maintain logical distinctness of the two.

This facility allows complicated template objects to be created, and then repeatedly copied without incurring a large storage overhead. An entire Ada library can be copied when a new project begins, and the old and new libraries will continue to share data blocks as long as the associated projects use without modification the common packages and subprograms.

3.2.4 Operations on Attributes

3.2.4.1 Operations on User-defined Attributes

(a) Specification. All database objects have a list of user-defined attributes, whose values are ASCII strings. The following primitives are used to set and retrieve these attributes:

```
procedure SET_ATTRIBUTE(NAME: in STRING; ATT_LABEL: in STRING;
    ATT_VALUE: in STRING);
     -- By setting an attribute value to the null STRING,
     --  the attribute is effectively deleted.
     -- Requires window giving USER_ATTRIB_WRITE (or WRITE).
     --  Will also fail if attribute is protected (see below).

function GET_ATTRIBUTE(NAME: in STRING; ATT_LABEL: in STRING)
    return STRING;
     -- Attribute value returned as null STRING if not
     --  previously SET.
     -- Requires window giving USER_ATTRIB_READ (or READ).

procedure PROTECT_ATTRIBUTE(NAME: in STRING; ATT_LABEL: in STRING;
    PROTECT: in BOOLEAN := TRUE);
     -- This procedure protects the specified user attribute
     --  from modification until called with parameter
     --  PROTECT => FALSE.
     -- Requires window giving SYSTEM_ATTRIB_MODIFY,
     --  as do other access control operations.
```

```
procedure SET_ALL_ATTRIBUTES(NAME: in STRING;
    ATT_VALUES: in STRING);
     -- All non-distinguishing attributes are
     --  set according to ATT_VALUES, which must be a
     --  parenthesized, comma-separated list of attribute
     --  label=>value pairs.
     -- Fails if any mentioned attribute is protected,
     --  or if window is insufficient.
     -- All attributes not explicitly mentioned, and not
     --  protected, are set to null.

function GET_ALL_ATTRIBUTES(NAME: in STRING) return STRING;
     -- Returns value of all non-null non-distinguishing
     --  user-defined attributes.
     -- STRING returned is in parenthesized labeled notation
     --  e.g. "(PURPOSE=>FUN,CHECK_LEVEL=>2)"
```

(b) Internal Representation and Algorithms. User-defined non-distinguishing attributes are stored as a simple ASCII string in the parenthesized labeled notation, with protected attributes flagged with an asterisk. Such a string is associated with each object that has any non-null attribute values. Additional blocks may be allocated for objects with large numbers of attributes.

SET_ATTRIBUTE and GET_ATTRIBUTE involve simple string manipulation of the label/value list. The time required to update the attribute list is linear in the number of attributes maintained for the object.

(c) Examples.

```
SET_ATTRIBUTE("TEST_FILE", "PURPOSE", "FUN");
SET_ATTRIBUTE("TEST_FILE", "CHECK_LEVEL", "1");
SET_ATTRIBUTE("XYZ", "PURPOSE", "FUN");
declare
     S: constant STRING := GET_ATTRIBUTE("XYZ", "CHECK_LEVEL");
          -- S is now the null STRING.
     AA: constant STRING := GET_ALL_ATTRIBUTES("TEST_FILE");
begin
     PUT_LINE(AA);
     -- Output will be: "(PURPOSE=>FUN,CHECK_LEVEL=>1)"
end;
```

3.2.4.2 Category Operations

(a) Specification. The category attribute is filled in automatically by CREATE, CREATE_COMPOSITE, CREATE_PCTX, and CREATE_WINDOW. More complex categories are specified using an Ada-like aggregate notation. The procedure SET_CATEGORY_ELEMENT or the program SET_CATEGORY is used to change the category provided by default at the time of creation.

The following primitives are available for inspecting or modifying the category specification of an object:

```
procedure SET_CATEGORY_ELEMENT(NAME: in STRING;
    CATEGORY_ELEMENT: in STRING; ELEMENT_VALUE: in STRING);
     -- This allows a user to change a single element
     --  of the category specification.
     -- Requires window giving SYSTEM_ATTRIB_MODIFY.

function GET_CATEGORY_ELEMENT(NAME: in STRING;
    CATEGORY_ELEMENT: in STRING) return STRING;
     -- This returns the value associated with a single
     --  element of the category specification.
     -- Requires window giving SYSTEM_ATTRIB_READ.

procedure SET_CATEGORY(NAME: in STRING;
    TEXT_FILE: in STRING);
     -- This program reads the specifed text file
     --  for lines of input, and uses SET_CATEGORY_ELEMENT
     --  to fill in the category attribute associated
     --  with the named object.

procedure GET_CATEGORY(NAME: in STRING; TEXT_FILE: in STRING);
     -- This program creates a text representation of the
     --  category of the named object on the given TEXT_FILE,
     --  using the primitive GET_CATEGORY_ELEMENT.
```

The category specification consists of a number of labeled category elements, each with a string value. The general form for the category specification for a simple object, as expected by SET_CATEGORY, is as follows:

```
(CATEGORY_CLASS => SIMPLE,
 CATEGORY_NAME    => arbitrary string,  -- Optional
 STORAGE_ORGANIZATION => < INDEXED | DIRECT >, -- Default is DIRECT
 BITS_PER_RECORD      => < 1 | 2 | ... >,  -- ELEMENT_TYPE'SIZE
 NON_DA =>
     (label1 [ => < value1 | (val11,val12,val13,...) |
                (low_numeric_val1 .. high_numeric_val1) >],
      label2 [ => < value2 | (val21,val22,val23,...) |
                (low_numeric_val2 .. high_numeric_val2) >],
      ...
     ), -- Optional, specifies required attribute labels and values
 ACCESS_CONTROL =>
     (capacity1 => (acc_rt11,acc_rt12,acc_rt13,...),
      capacity2 => (acc_rt21,acc_rt22,acc_rt23,...),
      ...
     ) -- Optional, specifies required access rights for
       --  specified capacities.
)
```

The strings CATEGORY_CLASS, CATEGORY_NAME, etc., are all names of category elements. In the above syntax, <choice1 | choice2 | ...> indicates the possible choices, [...] indicates an optional part.

When specifying the non-distinguishing attribute constraints (NON_DA element), each attribute may be limited to a single value (eg., value1 or value2 above), to a list of values (eg., val11, val12, val22 above), or to a numeric sequence of values (eg., "low_numeric_val1 .. high_numeric_val1" above). Either the low_numeric_val or the high_numeric_val may be "*" indicating negative and positive infinity, respectively. A constraint like "NON_DA => (NUMERIC_ID=>(*..*))" would specify that the non-distinguishing attribute NUMERIC_ID should be numeric, but with no other limits on its value.

The general form for a category specification for a composite object is as follows:

```
(CATEGORY_CLASS => COMPOSITE,
 CATEGORY_NAME  => arbitrary_string,  -- Optional
 NON_DA  => ..., -- As above for simple object NON_DA
 COMPONENT_DA =>  -- Required definition of component dist. attribs.
     (label1 [ => < (val11,val12,...) |
                 (low_num_val1 .. high_num_val1) >],
      label2 [ => < (val21,val22,...) |
                 (low_num_val2 .. high_num_val2) >],
      ...
     ),
 COMPONENT_CAPACITIES => (capacity1,capacity2,...),
         -- Optional, defines
         --  new capacities to be used by components
         -- of this object.
 COMPONENT_CATEGORIES =>  -- Optional constraints by partition
     (partition1 => category_spec1,
      partition2 => category_spec2,
      ...
     ),
 ACCESS_CONTROL =>
     -- Optional, specifies required composite object access
     --  rights and component capacities associated with designated
     --  external capacities.
     (capacity1 =>
            (acc_rt11 [ component_capacity11 ],
          acc_rt12 [ component_capacity12 ],
          ...
            ),
      capacity2 =>
            (acc_rt21 [ component_capacity21 ],
          acc_rt22 [ component_capacity22 ],
          ...
            ),
      ...
     ) -- end of ACCESS_CONTROL
) -- end of category specification
```

Each of the separate category elements (eg., CATEGORY_NAME, ACCESS_CONTROL) may be individually inspected or modified using GET_CATEGORY_ELEMENT or SET_CATEGORY_ELEMENT.

INTERMETRICS INCORPORATED • 733 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

(b) Internal Representation and Algorithms. The category attribute is stored on disk in a form designed to facilitate automatic constraint checking by the KAPSE. Category specifications are provided automatically for simple objects created with CREATE, and composite objects created with CREATE_COMPOSITE. These categories may later be adjusted element by element with SET_CATEGORY_ELEMENT, or may be completely replaced by SET_CATEGORY, which takes a text representation of the full category specification in the form given above, and fills in the category attribute accordingly.

No category element may be set in contradiction to the existing state of the attributes or content of the object. Similarly, the content and attributes may not be set in contradiction to the existing category specification. Some changes to the category may require that a category element be removed (set to null), the content or other attributes be modified, and then the category element be replaced with the desired new value.

Normally, a template object will be created using CREATE or CREATE_COMPOSITE followed by SET_CATEGORY, and then repeatedly COPY'ed to create new objects of the same category. As part of the delivered KAPSE, template objects are provided for such common composite objects as an Ada library, a user mailbox, and a typical user top-level directory.

(c) Examples. An example for a category specification for a simple object:

```
(CATEGORY_CLASS => SIMPLE, CATEGORY_NAME => ADA_SOURCE,
  STORAGE_ORGANIZATION => DIRECT,
  BITS_PER_RECORD => 8,
  NON_DA => (LANGUAGE=>ADA, CHECKING=>(NONE,SYNTAX,SEMANTICS))
)
```

An example for a category specification for a composite object:

```
(CATEGORY_CLASS => COMPOSITE, CATEGORY_NAME => ADA_LIBRARY,
  COMPONENT_DA =>
    (PER => (UNIT,COMPILATION,LINK)),
  COMPONENT_CATEGORIES =>
    ((PER=>UNIT) =>
     (CATEGORY_CLASS=>COMPOSITE,
       COMPONENT_DA=>(UNIT_NAME,SUBUNIT,UNIT_ID=>(0..4095))
     ),
     (PER=>COMPILATION) =>
     (CATEGORY_CLASS=>COMPOSITE,
       COMPONENT_DA=>(SEQUENCE_NUM=>(1..*))
     ),
     (PER=>LINK) =>
     (CATEGORY_CLASS=>COMPOSITE,
       COMPONENT_DA=>(LINKED_PROG_CTX_NAME)
     )
    )
)
```

3.2.4.3 Access Control

(a) Specification. The following primitives are available for manipulating the access control attribute:

```
procedure SET_CAPACITY_ACCESS(OBJNAME: in STRING;
    CAPACITY: in STRING; ACCESS_RTS: in STRING);
     -- Set list of access rights associated with given
     --  capacity.  Format for ACCESS_RTS is paren-
     --  thesized comma-separated list of access right
     --  names and optional component capacity.
     -- Requires window giving SYSTEM_ATTRIB_MODIFY on
     --  the specified object.

function GET_CAPACITY_ACCESS(OBJNAME: in STRING;
    CAPACITY: in STRING) return STRING;
     -- Return list of access rights associated with
     --  specified capacity for the designated object.
     -- Returned STRING is parenthesized comma-separated
     --  list of access right names and optional
     --  component capacity.
     -- Requires window giving SYSTEM_ATTRIB_READ (or READ)
     --  on the specified object.

function GET_CAPACITIES(OBJNAME: in STRING) return STRING;
     -- Return list of capacities with any access rights
     --  explicitly defined for this object.
     --  SYSTEM, OWNER, and INFERIOR not included unless
     --  explicitly limited to less than ALL.
     -- Requires SYSTEM_ATTRIB_READ or READ.
```

(b) Internal Representation and Algorithms. The access control attribute is represented by a simple table of capacity names and their associated access-right bit maps. Each bit map includes a bit for every access right meaningful for the object's category class. In the single case of the SELECT_COMPONENT access right, the internal component capacity name is also stored for each external capacity having this right.

Access to individual operation context objects associated with a private object is controlled by the access control attribute of the individual context objects. The private object as a whole limits access by restricting SELECT_COMPONENT at different internal capacities to specific external capacities. The operations then define the access for each of the internal capacities. A different internal capacity is defined for each meaningful combination of operations, with the name of the capacity suggesting the nature of the implied abstract role. Further control is also possible by limiting a window on the private object to a specific partition of the operations.

(c) Examples.

```
SET_CAPACITY_ACCESS("ALPHA", CAPACITY=>"WORLD",
    ACCESS_RTS=>"(READ,APPEND)");
     -- Give all users with WORLD window over simple
     --  object ALPHA access rights READ and APPEND.

SET_CAPACITY_ACCESS("BETA", CAPACITY=>"PROJECT",
    ACCESS_RTS=>"(COPY,SELECT_COMPONENT TESTER)");
     -- Give all users with PROJECT window over composite
     --  object BETA, right to COPY object, and access
     --  to its components in capacity TESTER.

PUT( GET_CAPACITY_ACCESS("ALPHA", CAPACITY=>"OWNER") );
     -- Will print "(ALL)" if never previously
     --  set otherwise.

PUT( GET_CAPACITIES("BETA") );
     -- Will print "(PROJECT)" if above SET_CAPACITY_ACCESS
     --  is the only one in effect for BETA.
```

3.2.4.4 History/Archiving Operations

(a) Specification.

```
with CALENDAR;  -- Defines type TIME.
Package HISTORY is

type HISTORY_CLASS is (SOURCE, DERIVED);
type HISTORY_REF(CLASS: HISTORY_CLASS := DERIVED) is limited private;
type HISTORY_REF_ARRAY is array(NATURAL range <>) of HISTORY_REF;

function GET_HISTORY_REF(NAME: in STRING) return HISTORY_REF;
     -- get current "STATE" of object.

procedure RECREATE(STATE: in HISTORY_REF(CLASS=>SOURCE);
    NAME: in STRING);
     -- Given the "STATE" of a source object, recreate
     --  its content and user attributes in a new database
     --  object with the given NAME.

procedure NEW_SOURCE_ARCHIVE(SOURCE_OBJ: in STRING);
     -- This creates a new source archive with
     --  SOURCE_OBJ as its state number one.
procedure OLD_SOURCE_ARCHIVE(SOURCE_OBJ: in STRING;
    STATE: in HISTORY_REF(CLASS=>SOURCE));
     -- This specifies that SOURCE_OBJ is a
     --  revision of STATE, and should be
     --  assigned to the same source archive.

function NUM_REFS(STATE: in HISTORY_REF) return INTEGER;
     -- Given a STATE, return count of number of other
     --  states directly derived from this state.
```

```
function GET_DERIVATIVES(STATE: in HISTORY_REF)
    return HISTORY_REF_ARRAY;
     -- Given a STATE, return list of other states
     --  directly derived from this state.
     -- If object is source object, list includes
     --  direct derivative source states in same archive.
     -- Other derivatives may not be known if reference
     --  listing turned off by user.

procedure SET_REFERENCE_LISTING(STATE: in HISTORY_REF;
    LISTING_ON: in BOOLEAN);
     -- This procedure sets reference listing on or off
     --  for the given STATE.  Reference counting is
     --  always performed.

function CHECK_REFERENCE_LISTING(STATE: in HISTORY_REF)
    return BOOLEAN;
     -- This function reports whether reference listing
     --  is currently on or off for the given STATE.

function GET_DIRECT_CONSTITUENTS(STATE: in HISTORY_REF)
    return HISTORY_REF_ARRAY;
     -- Given STATE, return list of states from which
     --  this state was directly derived.  If object is
     --  a source object, no more than one state is
     --  returned -- that of the direct predecessor
     --  to this state.

function GET_SOURCE_CONSTITUENTS(STATE: in HISTORY_REF)
    return HISTORY_REF_ARRAY;
     -- Given STATE, return list of source states from
     --  which this state was derived, directly or
     --  indirectly.  Derived object states are included
     --  in list only if their history was off-line
     --  and thus could not be traced immediately.

function GET_HISTORY_PARAMETERS(STATE: in HISTORY_REF)
    return STRING;
     -- For derived object state, return STRING
     --  with parameters provided at
     --  invocation of program producing STATE.
     -- For source object state, return list
     --  of the user attributes of the object
     --  at time of merge into archive.
     -- STRING is returned in (label=>value,...) format.

procedure HISTORY_ACTIVATE(STATE: in HISTORY_REF;
    TIME_LIMIT: in DURATION);
     -- This procedure requests that a particular history
     --  script or archive be activated (brought on-line).
     -- Depending on bulk-storage hardware, this may occur
     --  immediately or await operator attention, up to the
     --  specified TIME_LIMIT.
```

```
function HISTORY_ON_LINE(STATE: in HISTORY_REF) return BOOLEAN;
     -- This function returns TRUE if the referenced history
     --  script or archive is now active (on-line).

function HISTORY_TIME(STATE: in HISTORY_REF)
    return CALENDAR.TIME;
function HISTORY_MAKER(STATE: in HISTORY_REF)
    return STRING;
     -- The above two functions return the time/date and
     --  USER_NAME associated with the specified script
     --  or source archive STATE.
```

(b) Internal Representation and Algorithms. The history attribute of a database object consists of a HISTORY_REF; this uniquely identifies the state of the object. It refers to a source archive for a source object, or a program invocation script for a derived object (see 3.2.1.⁻). It includes an index to select one state from all those associated with the same source archive or script:

```
 private
type HUID is range 1..HUID_LIMIT;
     -- History unique identifier.
     --  These identifiers are assigned
     --  sequentially for each program
     --  execution resulting in database output,
     --  and for each source archive created.

type HISTORY_REF(CLASS: HISTORY_CLASS := DERIVED)
    is record
     ARCHIVE_SCRIPT: HUID;
     STATE_INDEX: INTEGER;
end record;

end HISTORY;
```

History unique identifiers are indexed by a central table within the KAPSE database. This table indicates whether the history source archive or script is on-line, or has been dumped to tape (bulk storage). If the referenced history is off-line, many of the above primitives will fail. The primitives HISTORY_ACTIVATE and HISTORY_ON_LINE may be used to affect or check the on-line status of a particular source archive or script.

All objects when initially created are treated as derived objects. The primitives NEW_SOURCE_ARCHIVE and OLD_SOURCE_ARCHIVE may be used to replace the history attribute reference to a program invocation script by a reference to a source archive. Source archives are used for maintaining multiple states of the same basic text, where the content itself is more important than the record of the program invocation script used to create the content. The date, time, and USER_NAME from the program invocation script are transferred to the source archive for each of its component states.

The source archive is stored in a form allowing the reconstruction of any of the component states in a single pass through it.

3.2.5 Other Database Operations

3.2.5.1 Synchronization

(a) Specification. The following primitives are used to effect synchronization among multiple Ada programs attempting to access overlapping parts of the database:

```
type RESERVE_MODE is (EXCLUSIVE_WRITE, READ_ONLY,
    SNAPSHOT_READ, SHARED_STREAM, SHARED_RANDOM);
     -- EXCLUSIVE_WRITE prevents all access except
     --  SNAPSHOT_READ.
     -- READ_ONLY prevents all write access.
     -- SNAPSHOT_READ never interferes, but may also be
     --  reading soon-to-be-obsolete data.
     -- SHARED_STREAM causes EXCLUSIVE_WRITE reservation
     --  only at the time of actual READ or WRITE.
     --  Stream READ always reads the first defined element of
     --  the object, and then advances FIRST to the next element.
     --  WRITE always appends a new element at the end of
     --  the object and advances LAST.
     -- SHARED_RANDOM causes a reserve (EXCLUSIVE_WRITE
     --  or READ_ONLY) only at the time of actual READ or WRITE.

procedure RESERVE(WINDOW_NAME: in STRING;
    MODE: in RESERVE_MODE; TIME_LIMIT: in DURATION);
     -- The target object of the named window is reserved
     --  according to the given RESERVE_MODE.
     -- If the RESERVE is not immediately possible
     --  due to a conflicting RESERVE, the caller is delayed
     --  up to the specified TIME_LIMIT, when a TIME_OUT
     --  exception will occur.

procedure RELEASE(WINDOW_NAME: in STRING);
     -- RELEASE after RESERVE for EXCLUSIVE_WRITE causes
     --  modifications made since the RESERVE to become
     --  permanent.
     -- RELEASE after READ_ONLY allows waiting writers to
     --  proceed to RESERVE.
     -- RELEASE after SNAPSHOT_READ throws away the logical
     --  COPY made for the purpose of uninterrupted reading.

procedure ABORT_RESERVE(WINDOW_NAME: in STRING);
     -- ABORT_RESERVE is equivalent to RELEASE for
     --  reserve modes READ_ONLY and SNAPSHOT_READ.
     -- After EXCLUSIVE_WRITE, an ABORT_RESERVE returns
     --  the reserved object or partition to its original
     --  pre-RESERVE state.
```

In addition, CREATE of a simple object, OPEN of a simple object, and OPEN_PARTITION result in implicit reserves. By default, OPEN for input only and OPEN_PARTITION do a SNAPSHOT_READ reserve. CREATE and OPEN for output do an EXCLUSIVE_WRITE reserve. The default reserve

may be overridden by additional information in the STRING passed to OPEN, providing for READ_ONLY reserve instead of SNAPSHOT, or selecting SHARED_STREAM or SHARED_RANDOM, in which case, an automatic RESERVE/RELEASE takes place around each READ and WRITE operation to the object (see 3.2.3.2).

(b) Internal Representation and Algorithms. After an Ada program performs a RESERVE, it may perform a sequence of operations using the reserved window without interference from other programs. When the sequence is complete, the program may RELEASE or ABORT_RESERVE. Each RESERVE starts by making a logical COPY of the reserved object. Modifications and accesses performed between RESERVE and RELEASE use this logical copy, preserving the integrity of the original object.

The KAPSE maintains an internal record of the full access path associated with each reserved or opened object. This internal record includes a dynamic join count (see 3.2.2 above), and is updated as appropriate due to concurrent operations on objects sharing all or part of the reserved object's access path.

The KAPSE implements RESERVE/RELEASE at a low level to allow efficient detection of conflicting reservations. When EXCLUSIVE_WRITE or READ_ONLY reservation of all or part of an object is requested, the KAPSE locates the smallest sub-tree of blocks fully enclosing the part to be reserved. If this sub-tree already contains a conflicting reserve, the new reserve is delayed up to the TIME_LIMIT. If not, the KAPSE marks the BLOCK_ID of the root of that sub-tree as reserved for either EXCLUSIVE_WRITE or READ_ONLY. In the case of write, it increments the reference count to produce a logical copy, and on first actual modification of the copy, splits the root and maintains a record of both BLOCK_IDs, one for the exclusive writer, and the other for SNAPSHOT readers.

3.2.5.2 Configuration Reporting and Management

(a) Specification. Configuration reporting and management are not separable from the rest of the KAPSE database facilities, but are, rather, integral to the reporting and management of attributes and partitions. The following KAPSE primitives, described in other sections of this document, are particularly relevant:

KAPSE Primitive	Section of this document
SET_ATTRIBUTE GET_ALL_ATTRIBUTES	3.2.4.1
CREATE_WINDOW	3.2.3.3
OPEN_PARTITION GET_NEXT_COMPONENT	3.2.3.2
SET_CATEGORY GET_CATEGERY	3.2.4.2
SET_CAPACITY_ACCESS GET_CAPACITY_ACCESS GET_CAPACITIES	3.2.4.3
GET_DIRECT_CONSTITUENTS GET_SOURCE_CONSTITUENTS GET_HISTORY_REFS	3.2.4.4

In addition to the above primitives, a standard program is provided to produce the configuration and attribute reports:

```
procedure LIST_PARTITION(PARTITION: in STRING := ".CURRENT_DATA.";
    ATTRIBUTES: in STRING := "");
     -- This program prints on the standard text
     --  output the distinguishing attributes
     --  (ie., names) of all of the components of the
     --  specified partition, as well as the requested
     --  non-distinguishing attributes, specified in
     --  the parameter ATTRIBUTES as a parenthesized,
     --  comma-separated list of attribute labels.
     -- If ATTRIBUTES is "*" then all non-null
     --  attributes of the components are printed.
     -- If ATTRIBUTES is null then no non-distinguishing
     --  attributes are printed.
     -- Notice that by default, the program lists the
     --  distinguishing attributes of all of the components
     --  of the partition implied by the .CURRENT_DATA
     --  window.
```

This program may be used to list attributes of:

1. The components of a composite object (ie., a configuration);

2. Some subset of the components, which satisfy a more complicated partition specification;

3. A single simple object.

(b) Internal Representation and Algorithms. The program LIST_PARTITION is implemented using the KAPSE primitives OPEN_PARTITION, GET_NEXT_COMPONENT, and GET_ALL_ATTRIBUTES.

(c) Examples.

```
SET_ALL_ATTRIBUTES("ALPHA", "(PURPOSE=>FUN,CHECK_LEVEL=>2)");
SET_ALL_ATTRIBUTES("BETA", "(PURPOSE=>WORK,CHECK_LEVEL=>2)");
SET_ALL_ATTRIBUTES("GAMMA", "(PURPOSE=>FUN)");

LIST_PARTITION("(CHECK_LEVEL=>2)", "(PURPOSE)");
    -- The following would appear on the standard output:
    --
    --  Partition (CHECK_LEVEL=>2)     Attributes (PURPOSE)
    --  ALPHA                          (PURPOSE=>FUN)
    --  BETA                           (PURPOSE=>WORK)

LIST_PARTITION("(PURPOSE=>FUN)", "(CHECK_LEVEL)");
    -- The following would appear:

    --  Partition (PURPOSE=>FUN)       Attributes (CHECK_LEVEL)
    --  ALPHA                          (CHECK_LEVEL=>2)
    --  GAMMA                          No CHECK_LEVEL

LIST_PARTITION();  -- Use the defaults
    -- The following might appear:

    --  Partition .CURRENT_DATA.
    --  ALPHA
    --  BETA
    --  DELTA
    --  GAMMA
    --  KAPPA

    -- Notice that all partitions are sorted in ASCII
    --  lexicographic order.
```

3.2.5.3 Backup and Recovery

An important design feature of the KAPSE is that backup and incremental recovery can be performed while the system is up and running. The tape (or bulk-storage) backup program begins by simply doing a SNAPSHOT_READ reserve of the root of the entire database. After that operation, the backup program may progress at its own pace through the hierarchy of objects, knowing that the data it reads reflects an internally consistent snapshot of the entire database.

(a) Specification. The following system programs are available for full and incremental backup, and incremental recovery:

```
Package BACKUP_RECOVERY is

procedure FULL_BACKUP(TIMESTAMP: out TIME_SEQ_NUMBER);
     -- This program copies a snapshot of the entire
     --  database to the tapes mounted by the operator.
     -- TIMESTAMP is the maximum time sequence number
     --  of any of the blocks transferred to tape.

procedure INCREMENTAL_BACKUP(BASE_LINE: in TIME_SEQ_NUMBER;
    TIMESTAMP: out TIME_SEQ_NUMBER);
     -- This program copies blocks to tape that have been
     --  modified since the BASE_LINE time sequence number.
     -- It also copies any block superior to a block that
     --  has been modified, to ensure that the copy on
     --  tape is a connected DAG (directed acyclic graph).

procedure RECOVERY(OLDNAME: in STRING; NEWNAME: in STRING;
    TIMESTAMP: in TIME_SEQ_NUMBER);
     -- This program attempts to re-create as NEWNAME
     --  the specified object as it was at the specified
     --  time sequence number.

 ...

end BACKUP_RECOVERY;
```

(b) Internal Representation and Algorithms. The KAPSE maintains an index of all backup tapes, indicating the range of time sequence numbers appearing on the tape. Each backup tape includes a header identifying its range. The rest of the tape is in a standard format with each block including its BLOCK_ID and reference count from when the block was dumped from disk. The blocks are topologically sorted before being dumped so that any element of the hierarchy on the tape may be located in a single sequential scan through the tape.

On recovery, the KAPSE instructs the operator to mount the appropriate incremental and full backup tapes, in order from latest to earliest, until the full content of the specified object has been reconstructed as of the requested time sequence number.

3.2.6 Program Invocation and Control

3.2.6.1 Program Context

(a) Specification. Each activation of a program has associated with it exactly one program context object. The following primitives are available to create new activations of a program by copying an executable program context template, and to suspend and restart execution of the program:

```
Package PROGRAM_INVOCATION is

function CALL_PROGRAM(PROGRAM_PATH: in STRING;
    PARAMETERS: in STRING; CONTEXT_NAME: in STRING :=
    ".SUB_PROGRAM_CONTEXT")
  return STRING;
    -- This function invokes an executable program
    --  context or command language script as
    --  though it were a sub-program of
    --  the calling program.
    -- PROGRAM_PATH is the access path to the program/script.
    -- PARAMETERS is a parenthesized, comma-
    --  separated list of parameters for the
    --  program, using positional or keyword
    --  notation (eg., "(A,B,EXTRA=>C)" ).
    -- The optional parameter CONTEXT_NAME specifies
    --  the LOCAL_NAME for the context object
    --  created for the called program.
    -- The returned STRING is a parenthesized
    --  comma-separated list of the out parameter
    --  values of the called program.
    -- If the called program is actually a function,
    --  the result is returned as though it were
    --  an out parameter labeled RETURN
    --  (eg., "(RETURN=>1423)" ).
    -- The current default text input and output of
    --  the calling program become the standard
    --  text input and output for the called program.
    -- All windows of the caller's context with
    --  INHERIT attribute equal to TRUE are copied
    --  into the sub-context created.
```

```
function PROGRAM_SEARCH(PROG_NAME: in STRING) return STRING;
     -- This function looks for an executable program
     --  context or command language script with
     --  name PROG_NAME in each of the composite
     --  objects specified in the caller's PROGRAM_SEARCH_LIST.
     -- The returned STRING is the full access path to
     --  the program context of script, ready to
     --  be passed to CALL_PROGRAM above.
     -- The PROGRAM_SEARCH_LIST is an attribute of the caller's
     --  program context object.  It is set using
     --  SET_ATTRIBUTE and specified as a parenthesized
     --  comma-separated list of composite object names.

procedure INITIATE_PROGRAM(PROGRAM_PATH: in STRING;
    PARAMETERS: in STRING; CONTEXT_NAME: in STRING;
    STD_INPUT: in TEXT_IO.IN_FILE;
    STD_OUTPUT: in TEXT_IO.OUT_FILE);
     -- This procedure invokes a program or
     --  script exactly like CALL_PROGRAM,
     --  except that the caller is not suspended
     --  until completion, and standard text
     --  input and output are specified
     --  explicitly.

function AWAIT_PROGRAM(CONTEXT_NAME: in STRING;
    TIME_LIMIT: in DURATION) return STRING;
     -- This function waits for the completion
     --  of the specified program context object,
     --  up to the specified TIME_LIMIT.
     -- The returned STRING is as in CALL_PROGRAM.

procedure SUSPEND_PROGRAM(NAME: in STRING);
     -- The program executing in the named context is stopped,
     --  allowing the state of the execution to be examined,
     --  or a debugger to be initiated to control or trace
     --  further execution of the program.
     -- Normal tasks of the program are made dormant, but
     --  the run-time system continues to respond to inter-
     --  program communication on channels zero and one.

procedure RESTART_PROGRAM(NAME: in STRING);
     -- The program associated with the named context is
     --  restarted. The program must have been previously
     --  initiated and then suspended.

procedure CREATE_PROGRAM_CONTEXT(CONTEXT_NAME: in STRING;
    PURE_PART: in STRING; IMPURE_PART: in STRING);
     -- A new program context object is created
     --  and initialized with the executable
     --  program image.  Additional windows or other
     --  objects may still be added to the program
     --  context before it is copied or initiated.
     -- This operation is normally performed only by the
     --  linker [I-5].
```

(b) Internal Representation and Algorithms. A program context is a composite object using a single component distinguishing attribute LOCAL_NAME and with certain standard windows and objects as components. In particular, every context includes a window CURRENT_DATA, which provides the main link to the permanent part of the database. The CURRENT_DATA window may be shifted to view other parts of the database using the CHANGE_VIEW (see 3.2.7.2 below). A running program has an implicit OWNER window on its program context.

The linker creates executable program context objects and by default deposits them in their associated Ada program library [I-5]. If the program is to be used by many users, it will be copied to a central repository of executable program context objects (eg., the TOOLS component of the root). When an executable program context is called or initiated, the KAPSE creates a private copy of it for this activation of the program. Unless otherwise specified, the copy is created as a component of the caller's context object.

When a command language script is called, the KAPSE invokes the standard command language processor and passes the name of the object containing the script as an additional parameter.

3.2.6.2 Parameter Passing

(a) Specification. Parameters are passed to a program context by CALL_PROGRAM and INITIATE_PROGRAM (see *** above) as a parenthesized comma-separated list using positional or keyword notation. For example:

```
CALL_PROGRAM("COMPILE", "(QSORT,MYLIB,OPTIM=>TIME)");
```

Internally, these parameters are passed as the value of an attribute of the created program context, labeled PARAMETERS. This attribute is then retrieved by the called program's preamble [I-5], by GET_ATTRIBUTE(".","PARAMETERS").

At the end of execution, values of out parameters are rewritten by the called program to the PARAMETERS attribute using SET_ATTRIBUTE, and are returned to the caller as the return string of CALL_PROGRAM or AWAIT_PROGRAM. If the called program is a function, the returned string is of the form "(RETURN=>return_value)." If the program ends due to an unhandled exception, the returned string will be "(EXCEPTION=>exception_id)."

The following function is defined to facilitate extracting a single parameter from the string returned by GET_ATTRIBUTE, CALL_PROGRAM, or AWAIT_PROGRAM:

```
function PICK_PARAM(PARAMETERS: in STRING; PARAM_NAME: in STRING;
    POSITION: in INTEGER := 0; DEFAULT: in STRING := "")
    return STRING;
     -- This function extracts the specified parameter from
     --  the given parameter string, as might be returned
     --  by GET_ATTRIBUTE(".", "PARAMETERS").
     -- PARAM_NAME may be null or POSITION may be zero,
     --  but not both.  The DEFAULT string is returned if
     --  no parameter is present in PARAMETERS at the
     --  designated POSITION or labeled by the
     --  specified PARAM_NAME.
```

(b) Internal Representation and Algorithms. The list of parameters is represented as the attribute PARAMETERS of the program context object. The function PICK_PARAM is provided to parse the parameter list, and does so by simply scanning through the PARAMETERS string supplied, looking for "PARAM_NAME =>" if PARAM_NAME is not null, or the unlabeled argument number POSITION. If neither is present, the supplied DEFAULT string is returned.

3.2.6.3 Private Object Operations

(a) Specification. The following primitives are available for creating and invoking private object operations:

```
function INVOKE_OPERATION(PRIV_OBJ: in STRING;
    OPERATION: in STRING; PARAMETERS: in STRING;
    TIME_LIMIT: in DURATION := 30.0) return STRING;
     -- This procedure attempts to invoke the specified
     --  operation context object (without copying it).
     -- If the operation is already active, the caller
     --  will be delayed up to the specified TIME_LIMIT
     --  (default 30 seconds).
     -- No windows are inherited from the calling program,
     --  but the KAPSE creates an INFERIOR window called
     --  .CALLER_CONTEXT for the operation to access
     --  the context of the calling program.
     -- The returned STRING is the out parameters
     --  or the return value of the operation.

procedure CREATE_PRIV_OBJ(NAME: in STRING);
     -- This procedure creates a new private object,
     --  by creating a composite object and a single
     --  DATA component.
     -- More often a user will copy an existing
     --  template private object than create a
     --  new one.
```

```
procedure ADD_OPERATION(PRIV_OBJ: in STRING;
    OPERATION_NAME: in STRING; PROG_CONTEXT: in STRING;
    OP_CAPACITY: in STRING := "OWNER");
     -- This procedure adds a new operation to an
     --  existing private object, by copying
     --  the designated PROG_CONTEXT object.
     -- The operation context is given a window
     --  on the DATA component of the specified
     --  OP_CAPACITY (by default OWNER).
     -- The window on the DATA component is called
     --  .CURRENT_DATA so that the operation may
     --  refer to it implicitly.
     -- The user controls access to the operation
     --  using the standard access control primitive
     --  SET_CAPACITY_ACCESS.  The only relavent
     --  access right is INVOKE.  The operation
     --  context object full access path is:
     --   PRIV_OBJ & "." & OPERATION_NAME
```

(b) Internal Representation and Algorithms. Private objects are implemented using normal composite objects, with a single component called DATA and a number of operation context-object components. Of the above subprograms, only INVOKE_OPERATION is actually a primitive. CREATE_PRIV_OBJ and ADD_OPERATION are both implementable directly in terms of existing composite object primitives:

```
procedure CREATE_PRIV_OBJ(NAME: in STRING) is
begin
     CREATE_COMPOSITE(NAME, COMPONENT_DA=>"OPERATION");

     SET_ATTRIBUTE(PRIV_OBJ, "NODE_LABEL",
         ATT_VALUE => "PRIVATE");
          -- Give object a NODE_LABEL so
          --  that windows created by ADD_OPERATION
          --  may use it as the common ancestor.

end CREATE_PRIV_OBJ;

procedure ADD_OPERATION(PRIV_OBJ: in STRING;
    OPERATION_NAME: in STRING; PROG_CONTEXT: in STRING;
    OP_CAPACITY: in STRING := "OWNER") is
     FULL_OP_PATH: constant STRING :=
          PRIV_OBJ & "." & OPERATION_NAME;
begin
     COPY(PROG_CONTEXT, FULL_OP_PATH);
          -- Copy to create operation context object.

     CREATE_WINDOW(NAME   => FULL_OP_PATH & ".CURRENT_DATA",
         TARGET           => PRIV_OBJ & ".DATA",
         COMMON_ANCESTOR => PRIV_OBJ,
         PARTITION        => "",
         CAPACITY         => OP_CAPACITY);
          -- Create .CURRENT_DATA window for
          --  operation context object.
end ADD_OPERATION;
```

INVOKE_OPERATION is implemented by attempting to reserve the designated operation context object for EXCLUSIVE_WRITE, and then starting it running with the given parameters and with a window .CALLER_CONTEXT referring to the caller's context. If the TIME_LIMIT expires while waiting for the reserve, the calling program receives a TIME_OUT exception.

3.2.6.4 Interprogram Communication

(a) Specification. Interprogram communication is performed by special operations on the associated program context objects. The form of the primitives is modeled after the Ada tasking primitives:

```
   function IPC_ACCEPT(CHANNEL_NUMBER: in INTEGER,
       TIME_LIMIT: in DURATION) return CALL_BLOCK;

   procedure IPC_END_RENDEZVOUS(RESULT: in CALL_BLOCK);

   procedure IPC_ENTRY_CALL(PCTX_NAME: in STRING,
       CHANNEL_NUMBER: in INTEGER, TIME_LIMIT: in DURATION,
       PARAMS: in out CALL_BLOCK);
        -- Requires a window giving PROGRAM_CTX_CONTROL
        --  on the specified program context object.

  end PROGRAM_INVOCATION;
```

(b) Internal Representation and Algorithms. Programs communicate over logical channels between them. Channel numbers zero and one are reserved for the Ada Run Time System and the Debug Support Routines. The use of other channels depends on the particular Ada program.

These interprogram communication primitives necessarily rely on the communicating programs agreeing on the format and interpretation of the CALL_BLOCK. From the KAPSE point of view, the CALL_BLOCK is just an array of bits. A TIME_LIMIT of zero results in a conditional ACCEPT or ENTRY call. A TIME_LIMIT of DURATION'LAST results in an effectively un-timed call. If a single program wishes to receive ENTRY calls on many channels simultaneously, it must execute the IPC_ACCEPT calls from separate Ada tasks.

3.2.6.5 Debugging and Control Interface

(a) Specification. The following procedures are available to a debugger for inspecting, controlling, and modifying a suspended program:

```
Package DEBUGGER_INTERFACE is

procedure SUSPEND_PROGRAM(NAME: in STRING);
procedure RESTART_PROGRAM(NAME: in STRING);
     -- These procedures are described in the
     --  section on Program Context (see 3.2.6.1).

procedure SET_CURRENT_DEBUGGED_CONTEXT(PROG_CTX: in STRING);
     -- This procedure is called once to specify
     --  which program context is being debugged,

procedure GET_PROGRAM_STATE(STATE: out PROGRAM_STATE);
     -- Retrieve the current state of the
     --  debugged program, including the program
     --  counter and stack pointer.

procedure CONTINUE(STATE: in out PROGRAM_STATE);
     -- Allow the debugged program to continue.
     -- This procedure returns when the debugged
     --  program reaches a breakpoint trap.

procedure SET_PROGRAM_DATA(ADDRESS: in ADDR_TYPE;
    DATA: in PACKED_BIT_VEC);
     -- Store the array of bits at the designated
     --  address in the debugged program.

procedure GET_PROGRAM_DATA(ADDRESS: in ADDR_TYPE;
    DATA: out PACKED_BIT_VEC);
     -- Retrieve into the array of bits data
     --  from the designated address in the
     --  debugged program.

procedure SET_ECP_BREAKPOINT(ADDRESS: in ADDR_TYPE;
    ON_OFF: in BOOLEAN);
     -- Activate or deactivate a breakpoint at
     --  the designated execution control point,
     --  according to ON_OFF.

procedure SET_EXCEPTION_BREAKPOINT(EXCEPTION_ID: in INTEGER;
    ON_OFF: in BOOLEAN);
     -- Associate or disassociate a breakpoint
     --  with the specified exception.

procedure SET_TRAPS(ALL_STATEMENTS, ALL_CALLS, ALL_RETURNS,
    ALL_EXCEPTIONS, UNHANDLED_EXCEPTIONS: in BOOLEAN);
     -- Associate or disassociate a breakpoint with
     --  the specified group of execution control
     --  points or exceptions.

end DEBUGGER_INTERFACE;
```

(b) Internal Representation and Algorithms. The above procedures are implemented using inter-program communication primitives. When a program is suspended, all of its normal tasks are made dormant, but a Debugger Support task of the standard Ada Run Time System remains responsive to inter-program communication on channel one. The Debugger Support task performs the requested operations on the debugger's behalf. See the Debugger B5 Specification [I-6] for a more complete discussion of the debugging interface.

3.2.6.6 Exception Handling

When an exception is raised explicitly within an Ada program, or implicitly due to a hardware-detected arithmetic or addressing violation, the run-time routine RAISE is invoked. If this routine determines that there is no active handler for the exception, and this is the main task of the program, the program is suspended at its current state (without "unwinding" the stack) and the upper-level program awaiting its completion (usually the command processor) is notified. At this point, the upper-level program may initiate a debugger to investigate the cause of the unhandled exception.

3.2.7 KAPSE User Interface

3.2.7.1 Overall User View of the Database

The overall structure of the database hierarchy is as follows:

```
                 ___________ROOT__________
                |           |     |         |
            SYSTEM       USERS    TOOLS    PROJECTS
```

The root composite object contains four components: SYSTEM, USERS, TOOLS, and PROJECTS. All of these components are themselves composite objects. The SYSTEM composite object contains objects of interest primarily to the system manager and certain maintenance tools (eg., backup, history indices, etc.).

The USERS composite object contains the top-level composite object (directory) for each user of the MAPSE. A particular component is selected by the user's USER_NAME (see LOGIN below).

The TOOLS composite object contains as components all of the standard MAPSE tools (and others added by a system manager). Each component is an executable program context object, or a command language script, selected by the distinguishing attribute TOOL_NAME.

The PROJECTS composite object has the component distinguishing attribute of PROJECT, and has initial components (PROJECT=>KAPSE) and (PROJECT=>MAPSE_TOOLS) for use by MAPSE developers.

3.2.7.2 LOGIN System

(a) Specification.

```
procedure LOGIN(USER_NAME: in STRING;
    USER_PASSWORD: in STRING);
     -- This is meant to be suggestive.  The user never
     --  explicitly calls this procedure.

procedure LOGOUT;

function CURRENT_USER_NAME return STRING;
     -- This function returns the current USER_NAME
     --  as specified to LOGIN.

procedure CHANGE_VIEW(PARTITION: in STRING);
     -- This procedure redefines the .CURRENT_DATA
     --  window to refer to the newly selected
     --  PARTITION.
     -- It is implemented using standard window
     --  operations (ie., CREATE_WINDOW)

procedure CHANGE_PASSWORD(PASSWORD: in STRING);
     -- This is meant to be suggestive.  Change password
     --  actually turns off echoing and requests the new
     --  password directly from the user's terminal.
     -- After confirmation, the new password is
     --  stored as the value of the USER_PASSWORD
     --  attribute of ".TOP_LEVEL_DATA." (see below).
```

(b) Internal Representation and Algorithms. When a user logs into the KAPSE, the LOGIN system requests a USER_NAME and a USER_PASSWORD (not echoed). The USER NAME is used to select a component from the USERS composite object (see 3.2.7.1 above). The password is encrypted using a non-invertible function and compared with the USER_PASSWORD attribute of this component. If the value matches, the component is taken to be the user's top-level directory (composite object), within which, by convention, exists a component named INITIAL_PROGRAM_CONTEXT, which is invoked on the user's behalf, with standard text input and output connected to the user's terminal. The INITIAL_PROGRAM_CONTEXT normally contains the executable program image for a full command language processor, but may contain a more restrictive program designed to provide a user with a more controlled environment (eg., text editing only).

From the INITIAL_PROGRAM_CONTEXT, the user may choose to CHANGE_VIEW to set up a different partition as the default. Alternatively, the user may choose to initiate a totally new program context at a different point in the database. For example, it might be that a project's library were implemented as a private object, with an operation MANAGE accessible only to users with a MANAGER window on it. The project manager could invoke this operation, which might be simply a command processor, but by so doing would prevent other conflicting access while (s)he performed a series of privileged operations on the DATA component of the project library private object.

No additional primitives are needed to manipulate the USERS composite object, or its components. Nevertheless, only users with an appropriate window on the USERS composite object can add new users to the system. Individual users may change their own USER_PASSWORD attribute, but not their USER_NAME.

When the MAPSE is initially installed, there is a single component of USERS named SYSTEM_MANAGER, with password SYSTEM. The SYSTEM_MANAGER composite object has an INITIAL_PROGRAM_CONTEXT with a SYSTEM window on the root of the entire database. The first action after installation should be to change the SYSTEM_MANAGER password.

Although sophisticated users or project managers could create for themselves an arbitrary INITIAL_PROGRAM_CONTEXT (limited of course by their access rights), most users will choose to follow the APSE standard for program contexts, which includes the following standard attributes and components:

Standard program-context attributes:

```
Attribute                      Default initial value
-------------------            ----------------------------
PROGRAM_SEARCH_LIST  =>   "(.TOOLS.,.CURRENT_DATA.)"

PARAMETERS           =>   "()"
                              -- No parameters to top-level
                              --  context.
                     =>   "(NAME=>ABC,COUNT=>3)"
                              -- Example of parameters to
                              --  lower-level context.

CONTEXT_STATE        =>   "INACTIVE"
                              -- State of program context
                              --  before activation or
                              --  after termination.
                     =>   "RUNNING"
                              -- State of program context
                              --  actively running.
                     =>   "SUSPENDED"
                              -- State of program context
                              --  suspended by user.
                              -- Context is waiting for
                              --  debugging commands,
                              --  restart, or termination.
```

Standard program-context components:

```
Component name               Category class
--------------               --------------
.CURRENT_DATA                Window
    -- Window on current partition, accessed by
    --  default when access path does not begin
    --  with a dot.

.TOP_LEVEL_DATA              Window
    -- Window on user's top-level composite object
    --  (directory).
```

```
.ROOT                       Window
    -- WORLD window on the root of the entire
    --  database.

.TOOLS                      Window
    -- WORLD window on the TOOLS composite object,
    --  in which are executable program context
    --  objects for the standard MAPSE tools.

.CALLER_CONTEXT             Window
    -- INFERIOR window on the program context of
    --  the invoking program.
    -- Not present in top-level context.

.PROGRAM_INITIAL_PURE       Simple object
    -- Pure part (code and constant data) of
    --  executable program image in format
    --  suitable for loading.

.PROGRAM_INITIAL_IMPURE     Simple object
    -- Initialization for impure part of
    --  image, in format suitable for loading.
    -- The executable program image is stored
    --  in the program context object by the
    --  linker [I-5].

.PROGRAM_SUSPEND_IMPURE     Simple object
    -- Impure part of program image saved by
    --  KAPSE when program suspended or aborted
    --  (including register values).

.PROGRAM_LINK_MAP           Simple object
.PROGRAM_LIBRARY            Window
    -- The LINK_MAP plus the PROGRAM_LIBRARY
    --  window provide sufficient information for
    --  a debugger to correctly inspect, control,
    --  and modify a suspended program.

.PROGRAM_HELP               Simple object
    -- This text file contains instructions and
    --  other documentation for the use of this
    --  program context.

.TERMINAL_INPUT             Simple object
.TERMINAL_OUTPUT            Simple object
    -- These two objects are managed by the
    --  KAPSE terminal handler.  Program
    --  I/O are connected to these text
    --  objects, with TERMINAL_INPUT lengthened,
    --  and TERMINAL_OUTPUT displayed
    --  by the terminal handler under
    --  keyboard control.
```

```
.OPEN_FILE_1                Reserved window
.OPEN_FILE_2                 "   "      "  "
.OPEN_FILE_3                 "   "      "  "
  etc...
    -- Each open file (or partition) handle
    --  is represented by a reserved window
    --  on the opened object (or partition).
    -- OPEN_FILE_1 and OPEN_FILE_2 are always
    --  associated with standard text input
    --  and output, respectively.

.SUB_PROGRAM_CONTEXT        Program context object
    -- This component is used by default to
    --  hold the program context for a program
    --  called as a sub-program.
    -- The PARAMETERS attribute of the context
    --  are the parameters to this sub-progam.
```

For efficiency, the root block of a running program context object, and other blocks on an LRU basis, are maintained in main memory. The program context captures in one object the information the KAPSE needs to know about a running Ada program.

3.2.7.3 User Accounting

(a) Specification. The following primitive exists to adjust budgets associated with objects:

```
procedure TRANSFER_BUDGETS(FROM: in STRING; TO: in STRING;
    DISK_AMOUNT: in INTEGER, PROCESSING_AMOUNT: in INTEGER);
      -- The designated number of budget units are
      --  transferred from one object to another.
      -- As usual, the names provided imply the windows on
      --  the object and thereby prevent unauthorized
      --  budget adjustments.
      -- The root of the entire database is assumed to
      --  have an unlimited budget, so that the system manager
      --  may dole out initial budgets from that object.
```

(b) Internal Representation and Algorithms. Associated with every object is a running total of the number of blocks that make up the object (including shared blocks), as well as a total of the number of processing resource units (CPU seconds) that have been used by components that are program context objects. These totals are updated on RELEASE of an EXCLUSIVE_WRITE reservation, and on program termination.

Along with the running totals, an object may have a disk block budget or a processing unit budget. When either running total exceeds the appropriate budget (if present), no further access or processing within the object may be initiated (already running programs are allowed to complete).

If an object does not have one of the budgets, it is limited only by the presence of a budget on some enclosing composite object. The budgets of the root of the entire database are both unlimited.

3.2.7.4 The Inter-User Mail System

(a) Specification. The following programs are available for sending and receiving inter-user mail:

```
procedure SEND_MAIL(TO_USER: in STRING; SUBJECT: in STRING;
    MESSAGE_OBJ: in STRING; MAIL_SEQ_NUM: out INTEGER);
     -- This program sends mail to the designated user.
     --  The program constructs a path as
     --  ".ROOT.USERS." & TO_USER & ".MAILBOX"
     --  and attempts to invoke the operation
     --  SEND on this private object.
     --  If the caller lacks sufficient access
     --  rights through this path, SEND_MAIL will fail.
     -- In addition, this requires a window allowing
     --  COPY of the MESSAGE_OBJ.
     -- The returned MAIL_SEQ_NUM may be used to check
     --  if the mail has been read.

function SEND_MAIL_CHECK(MAIL_SEQ_NUM: in INTEGER)
    return BOOLEAN;
     -- This function indicates whether the message
     --  with the specified MAIL_SEQ_NUM has been
     --  read.
     -- This function simply fails if the message
     --  was not sent by the caller.

function CHECK_MAIL return INTEGER;
     -- This function returns a count of the number
     --  of message objects in the user's MAILBOX.
     -- The path to the mailbox is assumed to be
     --  ".TOP_LEVEL_DATA.MAILBOX"

procedure READ_MAIL(MESSAGE_OBJ: in STRING);
     -- The next message in the user's mailbox is
     --  is copied into the specified MESSAGE_OBJ.
     -- The following non-distinguishing attributes
     --  of this MESSAGE_OBJ will have appropriate values:
     --  FROM          =>   USER_NAME of SENDer,
     --  SUBJECT       =>   SUBJECT as specified by SENDer,
     --  MAIL_SEQ_NUM =>    Mail sequence number of this
     --                            message.
```

(b) Internal Representation and Algorithms. Mail is implemented using private object operations. When a new user is added to the system, the system manager creates a private object called MAILBOX in the user's top-level composite object by copying the standard system mailbox template. Each of the MAIL subprograms given above simply invoke the appropriate operation of a mailbox private object.

For example, SEND_MAIL could be written in Ada as follows:

```
procedure SEND_MAIL(TO_USER: in STRING; SUBJECT: in STRING;
    MESSAGE_OBJ: in STRING; MAIL_SEQ_NUM: out INTEGER) is
     MAIL_PATH: constant STRING :=
          ".ROOT.USERS." & TO_USER & ".MAILBOX";
     MAIL_PARAMS: constant STRING :=
          "(FROM_USER=>"    & CURRENT_USER_NAME() &
          ",SUBJECT=>"      & SUBJECT             &
          ",MESSAGE_OBJ=>"  & MESSAGE_OBJ & ")";
begin
     MAIL_SEQ_NUM :=
       INTEGER'VALUE(PICK_PARAM(
         INVOKE_OPERATION(
          PRIV_OBJ     =>  MAIL_PATH,
          OPERATION    =>  "SEND",
          PARAMETERS   =>  MAIL_PARAMS
          ),
         "MAIL_SEQ_NUM"
       ));

end SEND_MAIL;
```

3.2.7.5 User Terminal Handling

(a) Specification. The KAPSE provides a standard set of terminal control facilities, directly available to the interactive MAPSE user:

ASCII Key Code	Terminal Control Function
Control-S (XOFF)	Stop terminal output. Enter Scroll Control Mode. (see below)
Control-Q (XON)	Exit Scroll Control Mode. Re-start terminal output.
Control-C (ETX) or BREAK	Interrupt running program, Give control to program catching input interrupts.
Control-H (Backspace)	Erase previous entered character.
Control-X (Cancel)	Erase entire line entered.

Scroll Control Mode is provided for terminal users to review output which has gone off the screen of a video terminal, or was illegible or lost from the printout of a hardcopy terminal.

In Scroll Control Mode, the terminal handler recognizes the following small number of commands:

ASCII Key Code	Scroll Control Mode Function
B	"Back" -- Scroll the screen backward half of a screenful, or simply retype the previous line on a hardcopy terminal.
digit B	Go back specified number of half screens or lines, and redisplay.
F	"Forward" -- Scroll the screen forward half of a screenful, or simply retype the next line which had been typed on a hardcopy terminal.
digit F	Go forward specified number of half screens or lines, and redisplay.
Control-C or BREAK	Exit Scroll Control Mode and Interrupt program as above.
Control-Q	Exit Scroll Control Mode, return to display of current terminal output.

On terminals without normal ASCII keyboards, the user may define alternate character sequences to replace the ASCII control characters, using the SET_INPUT_INFO primitive (see 3.2.8.4 below). On half-duplex systems, all control characters (or sequences) must be preceded by an attention key, and terminated by the end-of-line character so that characters are received by the KAPSE.

(b) Internal Representation and Algorithms. Scroll Control Mode is possible because all terminal output is saved temporarily in the program context component .TERMINAL_OUTPUT. At the end of program execution, this component may be saved if the output is considered valuable.

In addition, all terminal input to a program is stored temporarily in the program context component .TERMINAL_INPUT, so that historical records of program invocation can be complete. At the end of program execution, a user may copy the .TERMINAL_INPUT component into a more permanent part of the database to avoid having to re-enter the same input if the program is re-run at a later time. From the point of view of history, .TERMINAL_INPUT is treated as a source object.

3.2.8 Ada Run Time System and High-Level I/O

3.2.8.1 Ada Tasking Model

(a) Specification. Part of the KAPSE design is a model for the Ada tasking run-time system written in Ada. See Appendix 10.2 for complete details of the model. Listed below are the primitives accessible to the compiled code to implement the various tasking constructs:

```
Package ADA_RUN_TIME is

procedure SET_DELAY(AMOUNT: in DURATION);

procedure SIMPLE_ACCEPT(ENTRY_NUM: in INTEGER);

procedure ENTRY_CALL(TSK: in TCB_PTR; ENTRY_NUM: in INTEGER;
     TIME_LIMIT: in DURATION);

procedure SET_OPEN(ENTRY_NUM: in INTEGER; SLCT_INDEX: in INTEGER);

procedure READY_TO_TERMINATE;

procedure SELECT_CALLER(TIME_LIMIT: in DURATION);

procedure ABORT_TASK(TSK: in TCB_PTR);

procedure TERMINATE;

procedure CREATE_TASK(TSK: out TCB_PTR; PRIO: in PRIORITY;
     NUM_ENTRIES: in INTEGER);
     -- Create a new TCB and add it to the
     --  current scope's initialization queue.

procedure INITIATE_TASKS;
     -- Initiate all tasks on the current
     --  scope's initialization queue.
procedure RAISE_FAILURE(TSK: in TCB_PTR);
```

(b) Internal Representation and Algorithms. A task control block (TCB) is allocated for each active task. A globally accessible variable contains a pointer to the current running task TCB. All other tasks are on either an initialization queue, a runnable task queue, an entry call queue, or a rendezvous stack.

3.2.8.2 Storage Management

(a) Specification. Two primitives are provided by the KAPSE to control the total storage allocated to a single program:

```
procedure GET_STORAGE(AMOUNT:  in NATURAL; WHERE:  out ADDR_TYPE);
     --AMOUNT is given in STORAGE_UNITs.

procedure FREE_STORAGE(AMOUNT:  in NATURAL; WHERE:  in ADDR_TYPE);

end ADA_RUN_TIME;
```

(b) Internal Representation and Algorithms. The KAPSE keeps track of storage allocated to the various running Ada programs, allowing them to dynamically increase and decrease their allocation as execution progresses. The host system may limit total allocation, and may require that actual allocation be fixed for the lifetime of a program, so that the dynamic allocation only affects storage already committed to the program.

3.2.8.3 Package INPUT OUTPUT

(a) Specification. Package INPUT_OUTPUT is implemented according to the specification in the Ada LRM [G-1, section 14.1].

(b) Internal Representation and Algorithms. Internally, all operations are converted to operations on bit arrays, allowing arbitrary types of objects to be handled. The conversion to standard types is made within the generic body of the package, while the bulk of the processing is done in a non-generic package to avoid multiple instantiations.

3.2.8.4 Package TEXT IO, Interactive/Terminal I/O Extensions

(a) Specification. Package TEXT_IO [G-1, 14.3] is extended to include additional operations to facilitate interactive text I/O. All operations succeed on normal disk text files, although they may not have any effect.

```
Package TEXT_IO is

package CHARACTER_IO is new INPUT_OUTPUT(CHARACTER);

type IN_FILE is new CHARACTER_IO.IN_FILE;
type OUT_FILE is new CHARACTER_IO.OUT_FILE;

 ... -- As in Ada LRM
```

```
procedure SET_ECHO(INPUT: in IN_FILE; OUTPUT: in OUT_FILE);
     -- Sets cursor and echoing of INPUT at current
     --  line and column of output.  Each character GET from
     --  INPUT advances the column of both the INPUT and
     --  the OUTPUT files (although the column numbers will
     --  not necessarily be the same).

procedure NO_ECHO(INPUT: in IN_FILE);
procedure NO_ECHO(OUTPUT: in OUT_FILE);
     -- Either of these calls will break any
     --  echoing association.

procedure SET_LINE_LENGTH(FILE: in OUT_FILE; N: in INTEGER);
     -- As in Ada LRM 14.3.2.

procedure SET_COL(FILE: in OUT_FILE; TO: in NATURAL);
     -- As in Ada LRM 14.3.2.

procedure SET_LINE(FILE: in OUT_FILE; TO: in NATURAL);
     -- Only allowed if a fixed LINE_LENGTH has been
     --  specified for the output file.
     -- This procedure is used to provide random access
     --  terminal screen output.

procedure GET_OUTPUT_INFO(FILE: in OUT_FILE;
    INFO: out OUTPUT_INFO_BLOCK);

procedure SET_OUTPUT_INFO(FILE: in OUT_FILE;
    INFO: in OUTPUT_INFO_BLOCK);
     -- The OUTPUT_INFO_BLOCK retains information such as
     --  the terminal's screen height and width (zero height
     --  indicates hard copy, zero width indicates OUT_FILE
     --  is not associated with a physical terminal).

procedure GET_INPUT_INFO(FILE: in IN_FILE;
    INFO: out INPUT_INFO_BLOCK);

procedure SET_INPUT_INFO(FILE: in IN_FILE;
    INFO: in INPUT_INFO_BLOCK);
     -- The INPUT_INFO_BLOCK retains information such as
     --  the specific keyboard control characters used to
     --  control the various terminal handling functions.
     -- In addition, the INPUT_INFO_BLOCK records
     --  which characters cause program wakeup when
     --  typed (others are buffered up and a control
     --  character may be used to delete them
     --  before they are received by a program).

 ...

end TEXT_IO;
```

(b) Internal Representation and Algorithms. All terminal output is actually written to a temporary file in the program's context object. All operations such as SET_LINE and SET_COL are in terms of this temporary file. The terminal handler normally keeps the last line of this temporary file as the last line on the screen. However, the user may choose to scroll backward to see previous lines of output, or to simply hold the screen image at a particular line (see 3.2.7.5). When echoing is set, the terminal handler makes sure that the current LINE and COL of the output are on the screen before setting the cursor there and requesting input on the associated IN_FILE.

3.2.8.5 Package FORMATTED IO

(a) Specification. Along with the above extensions to TEXT_IO, the KAPSE defines a FORMATTED_IO package to provide the facilities of Fortran-like FORMAT I/O:

```
Package FORMATTED_IO is

type FORMAT is private;

function CONV_FMT(FMT: in STRING) return FORMAT;
     -- Given a STRING in Fortran FORMAT syntax, check
     --  the correctness of the syntax and compress to
     --  facilitate further use.

procedure FWRITE(FILE: in TEXT_IO.OUT_FILE; FMT: in FORMAT);
     -- Start output using the given (compressed) FORMAT.

procedure FPUT(ITEM: in STRING);
     -- This uses the "Aw" format.
procedure FPUT(ITEM: in FLOAT);
     -- This typically uses "Fw.d" formats.
procedure FPUT(ITEM: in INTEGER);
     -- This typically uses the "Iw" format.
     -- Continue output, using the next format specifier
     --  from the format specified in the most recent FWRITE call.
     -- The user may choose to further overload FPUT by writing
     --  versions that take a sequence of INTEGERS or FLOATS or
     --  some useful combination.

procedure FEND;
     -- Terminate output, force characters out to file.
procedure FREAD(FILE: in TEXT_IO.IN_FILE; FMT: in FORMAT);
     -- Start input using the given (compressed) FORMAT.
```

```
   procedure FGET(ITEM: out FLOAT);
        -- This typically uses the "Fw.d" format.
   procedure FGET(ITEM: out INTEGER);
        -- This typically uses the "Iw" format.
        -- Continue input, using the next format specifier from
        --  the FORMAT specified in the most recent FREAD call.
        -- The user may choose to further overload FGET by writing
        --  versions that take a sequence of INTEGERS or FLOATS
        --  or some other useful combination.

    ...

   end FORMATTED_IO;
```

(b) Internal Representation and Algorithms. The package FORMATTED_IO is implemented in Ada, using package TEXT_IO and package INPUT_OUTPUT, ensuring that it is easily transportable to other Ada installations.

(c) Examples.

```
   declare
        F1: constant FORMAT := CONV_FMT( "2I3, F8.2" );
        I,J,K: INTEGER := 5;
        Z: FLOAT := 3.22;
   begin
        FWRITE(FILE, F1);
        FPUT(I+J); FPUT(25); FPUT(Z); FEND;

        FWRITE(FILE,CONV_FMT(" 'The Answer is ',I6//"));
        FPUT(K*127); FEND;
   end;
```

3.2.9 KAPSE/Host Interface -- VM/370 and OS/32

3.2.9.1 Overall Architecture

The overall architecture of the KAPSE/Host interface is a number of independently executing Ada programs running concurrently on the host machine. Each independent Ada program has its own run-time system, including an Ada task scheduler. The host system provides the timesharing and swapping of the independent programs.

One of the Ada programs is special -- the Data Base Manager. This program insulates the rest of the programs from most of the idiosyncracies of the host system facilities. As far as is possible on the particular host, the other programs are prevented from accessing host facilities directly, thus ensuring that the KAPSE Database is not corrupted.

(a) IBM VM/370. This overall logical architecture is mapped onto the VM/370 system by providing each independent user with a separate virtual machine (see IBM VM/370 documentation [N-1]). After LOGIN, the virtual machine has a single program running in it: the user's command language processor. The additional programs initiated by the command processor share this same virtual machine. The multiple programs within a single virtual machine are managed by the User VM Manager running in supervisor mode in the virtual machine.

In addition to the user virtual machines, the KAPSE requires that its own virtual machine be initialized with the Central Data Base Manager (CDBM). The CDBM initiates all physical disk I/O and includes the central buffer cache. The individual User VM Managers handle terminal I/O, in cooperation with the CDBM. All communication between virtual machines is performed using the Virtual Machine Communication Facility (VMCF), a high-band-width memory-to-memory data path provided by the VM/370 Control Program [N-1].

(b) Perkin-Elmer OS/32. The same overall logical architecture is mapped differently onto the OS/32 system, by providing each independent executing Ada program with its own OS/32 task. User programs execute in a mode whereby the only OS/32 SVC 6 system calls they can perform are inter-task communication. They are not permitted to directly stop, start, or otherwise interfere with other tasks (NOCON mode -- see OS/32 Programmer's Manual [N-2]).

The Data Base Manager (DBM) runs in its own OS/32 task, with access to all OS/32 system calls. It initiates all physical I/O, including terminal and disk, and thereby can optimize physical disk access and provide the central buffer cache. All OS/32 tasks communicate using the standard OS/32 inter-task communication primitives, a memory-to-memory queue-based data path [N-2].

3.2.9.2 Physical Disk I/O

(a) Specification. The following low-level subprograms are implemented for each host, to provide physical disk I/O and allocation:

```
Package KAPSE_HOST_INTERFACE is

function ALLOCATE_BLOCK(PREDECESSOR: in BLOCK_ID)
    return BLOCK_ID;
  -- This function allocates one physical block, and
  --  initializes its reference count to one.
  -- If PREDECESSOR is non-zero, ALLOCATE_BLOCK
  --  attempts to allocate a block as close as
  --  possible to the optimal separation from it,
  --  so that later sequential access should be able to
  --  get successive blocks without missing revolutions.
```

```
procedure INCREMENT_BLOCK_REF(BLK: in BLOCK_ID);
     -- This procedure increments the reference count to
     --  the designated block.

procedure DECREMENT_BLOCK_REF(BLK: in BLOCK_ID);
     -- This procedure decrements the reference count to
     --  the designated block.  If the count reaches zero,
     --  the block is made available for future
     --  ALLOCATE_BLOCK calls.

type BUFFER_DATA(MAX_NUM_BITS: INTEGER) is record
     NUM_BITS: 0..MAX_NUM_BITS;
          --  Current number of bits of DATA.
     DATA: PACKED_BIT_VEC(1..MAX_NUM_BITS);
end record;

type BUFFER_DATA_PTR is access BUFFER_DATA;
     -- BUFFER_DATA is used for holding
     --  actual data of a block.

task type BUFFER is
     entry FILL(DATA: in BUFFER_DATA_PTR);
     entry DRAIN(D^TA: out BUFFER_DATA_PTR);
end BUFFER;

type BUFFER_PTR is access BUFFER;
     -- BUFFER is used to synchronize access to
     --  a buffer of data.
     -- FILL is accepted only when the BUFFER is
     --  empty, and leaves it full.
     -- DRAIN is accepted only when the BUFFER is
     --  full, and leaves it empty.


procedure READ_BLOCK(BLK: in BLOCK_ID; DATA: in BUFFER_PTR);
     -- This procedure reads in the block
     --  designated by BLK.
     -- READ_BLOCK returns immediately;
     --  the data is filled in asynchronously.

procedure WRITE_BLOCK(BLK: in BLOCK_ID; DATA: in BUFFER_PTR);
     -- This procedure writes out the block
     --  designated by BLK.
     -- WRITE_BLOCK returns immediately;
     --  the data is drained asynchronously.
```

(b) Internal Representation and Algorithms --VM/370. The Central Data Base Manager virtual machine is assigned a number of virtual mini-disks within the VM/370 Directory. Each of these mini-disks consists of a number of cylinders, with each cylinder holding a number of the KAPSE fixed-size blocks. The BLOCK_ID returned after block allocation identifies the mini-disk, the cylinder, and the block within cylinder.

Blocks are allocated so that sequential blocks are in the same cylinder, if possible, with a separation from the predecessor block determined by the physical characteristics of the device type of the mini-disk. The logically sequential blocks of an object are allocated non-contiguously to allow for the delays associated with a time-sharing environment, which prevent a user program from processing data as fast as the disk could provide it.

The reference counts are maintained in their own disk blocks, separately from the data blocks. They may be updated without rewriting the data of the block itself. The reference counts are scanned to locate a free block in the predecessor's cylinder, with the appropriate separation. Recently accessed reference count blocks are cached in main memory to speed this process.

(c) Internal Representation and Algorithms --OS/32. The OS/32 Data Base Manager task obtains disk storage by creating contiguous OS/32 files with a consistent naming scheme. The files are then assigned to the DBM with exclusive read/write, thereby preventing other OS/32 tasks from corrupting the data. After creating such a file, it is treated much like the VM/370 mini-disk, with reference counts and data placed in separate areas of the file.

3.2.9.3 Terminal I/O

(a) Specification. The following primitives are available to the KAPSE for terminal input/output:

```
procedure READ_TERMINAL(TERM: in INTEGER; ECHO: in BOOLEAN;
    DATA: in BUFFER_PTR; MAX_CHARS: in INTEGER);
     -- This procedure sets up a buffer for characters
     --  to be read from the specified terminal,
     --  with or without echoing.
     -- The buffer will be filled when the MAX_CHARS limit
     --  is reached, or when any ASCII control
     --  character is typed (including DEL).
     -- NUM_BITS of the associated BUFFER_DATA
     --  indicates actual number of characters accepted.
     -- With MAX_CHARS => 1, the buffer is filled as
     --  soon as the next character is typed.
     -- ASCII control characters are never echoed
     --  by READ_TERMINAL, independent of ECHO.
```

```
procedure WRITE_TERMINAL(TERM: in INTEGER;
    DATA: in BUFFER_PTR)
     -- This procedure writes characters to the
     --  specified terminal.
     -- DATA must have been filled in previously,
     --  and will be drained asynchronously.

procedure SET_TERMINAL_INFO(TERM: in INTEGER;
    INFO: in TERMINAL_INFO_BLOCK);

procedure GET_TERMINAL_INFO(TERM: in INTEGER;
    INFO: out TERMINAL_INFO_BLOCK);
     -- These procedures pass along information
     --  between the host terminal device driver
     --  and the KAPSE terminal handler.
     -- In the case of hard-wired terminals, the host
     --  may know the characteristics of the
     --  terminal.  For dial-up terminals, the user
     --  must in general specify the appropriate
     --  information explicitly via SET_INPUT_INFO
     --  and SET_OUTPUT_INFO (see 3.2.8.4 above),
     --  which the KAPSE will then digest and send
     --  along via SET_TERMINAL_INFO.
```

(b) Internal Representation and Algorithms --VM/370. Each User VM Manager controls input and output for its own associated terminal, using the virtual interface provided by the VM/370 Control Program (CP). The Central Data Base Manager informs the User VM Manager which file handles of the user programs refer to the terminal, allowing the User VM Manager to intercept reads and writes and handle the requests directly.

(c) Internal Representation and Algorithms --OS/32. The Data Base Manager task on OS/32 handles all terminal I/O for the KAPSE. Individual user tasks need not be rolled in for echoing to proceed, and character and line deletion to be processed.

For each user a separate Ada task within the Data Base Manager handles the terminal. When an input buffer is complete, the waiting user OS/32 task is activated by sending it a message containing the characters.

3.2.9.4 Device Input/Output and Import/Export

(a) Specification. Device objects (see CREATE_DEVICE_OBJ above) are used as the access points for device I/O and import and export. Because only a system manager may create device objects, the correct syntax for HOST_DEVICE_NAME need not be known to the normal user, and may be host-dependent.

The following primitives exist to read or write host files or phsical I/O devices:

```
type FILE_MODE is (IN_MODE, INOUT_MODE, OUT_MODE);
type DEVICE_HANDLE is private;

OPEN_DEVICE(DH: in out DEVICE_HANDLE;
    HOST_DEVICE_NAME: in STRING; MODE: in FILE_MODE);

READ_DEVICE(DH: in DEVICE_HANDLE; DATA: in BUFFER_PTR);

WRITE_DEVICE(DH: in DEVICE_HANDLE; DATA: in BUFFER_PTR);

CLOSE_DEVICE(DH: in out DEVICE_HANDLE);
     -- Whenever a user reads or writes a device
     --  object, the KAPSE retrieves the HOST_DEVICE_NAME
     --  stored when the device object was created,
     --  and passes the request off to these KAPSE/Host
     --  interface procedures.

SET_DEVICE_INFO(DH: in DEVICE_HANDLE;
    INFO: in DEVICE_INFO_BLOCK);

GET_DEVICE_INFO(DH: in DEVICE_HANDLE;
    INFO: out DEVICE_INFO_BLOCK);
     -- A certain amount of device control and status
     --  information may be set and retrieved using
     --  these calls. These are externally accessible
     --  as KAPSE calls SET_FILE_INFO and GET_FILE_INFO.
```

(b) Internal Representation and Algorithms --VM/370. On the VM/370 the HOST_DEVICE_NAME implies the virtual device address and device type. Using commands to VM/370 CP, a user or operator can connect what appears to be a virtual punch on one VM to be a virtual card reader on some other VM. In this way, export/import can be with actual devices, or files on other operating systems.

(c) Internal Representation and Algorithms --OS/32. On OS/32 the HOST_DEVICE_NAME implies the physical device mnemonic, or the volume and file name of the host file.

3.2.9.5 Ada Tasking Support

The KAPSE uses Ada tasking constructs to accomplish the management of multiple concurrent database and inter-program communication requests. Each Ada program has run-time routines to provide multi-tasking, but requires additional support to provide time-based scheduling. The support of the Ada tasking run time routines is thus an important part of the KAPSE/Host interface.

(a) Specification.

```
procedure SET_INTERRUPT_SERVICE(ADDR: in ADDR_TYPE);
     -- This routine is called by the
     --  Ada run time system to specify
     --  the routine which will handle all
     --  (pseudo) interrupts from the host.

type INTERRUPT is (CLOCK, MESSAGE);
procedure INTERRUPT_SERVICE_ROUTINE(KIND: in INTERRUPT;
    DATA: in ADDR_TYPE);
     -- This is the spec for a typical interrupt
     --  service routine.
     -- When the routine is called, the parameters
     --  indicate the kind of interrupt and where
     --  any associated data reside.
     -- The address of this routine is passed
     --  to SET_INTERRUPT_SERVICE.

procedure GET_TIME_OF_DAY(TIME: out CALENDAR.TIME);

procedure SET_TIMER(HOW_LONG: in DURATION);
     -- Request that a timer interrupt be
     --  generated after the specified duration.
```

(b) Internal Representation and Algorithms --VM/370. On the VM/370, the Control Program handles time-sharing and paging among separate virtual machines, while the User VM Managers handle time-slicing among the multiple Ada programs within a single VM.

A User VM Manager provides an Ada program with a pseudo interrupt when its timer goes off, or when a message is received. All timers are based on real time rather than virtual time, using the Set Clock Comparator instruction [N-1].

(c) Internal Representation and Algorithms --OS/32. On OS/32, the task queue facility is used to implement SET_INTERRUPT_SERVICE, and the timer management system calls are used to implement SET_TIMER [N-2].

3.2.9.6 Program Loading and Initiation

(a) Specification. The following procedure is provided to interface to the host program loading and initiation facilities:

```
procedure LOAD_PROGRAM(PURE_PART: in STRING;
    IMPURE_PART: in STRING; ID: out PROGRAM_ID);
     -- This procedure loads and initiates the designated
     --  program.  The returned PROGRAM_ID may
     --  be used later to communicate with the
     --  program.
     -- The PURE_PART and IMPURE_PART are simple
     --  objects in a form suitable for loading
     --  by the host system.  The history attribute
     --  of the PURE_PART uniquely identifies the
     --  state of its content, and the implementation
     --  may attempt to share code for multiple
     --  programs using the same PURE_PART.
```

The historical age of the PURE_PART is used as an indication of its MAPSE longevity. Extra effort will be made to share PURE_PART code which is indicated to be sufficiently "established."

(b) Internal Representation and Algorithms --VM/370. Programs are loaded by transferring the code and data images via VMCF, and then relocated to an available location within the user's virtual machine by the User VM Manager.

A limited number of named discontiguous shared segments are created and allocated when the KAPSE is installed, for the purpose of holding images of code used simultaneously by separate VMs. When sharing is warranted, the Central Data Base Manager will copy the pure part into an available shared segment, by first turning off protection, copying into the segment, and then turning it back on. It then informs the appropriate User VM Managers the name of the shared segment [N-1].

(c) Internal Representation and Algorithms --OS/32. Unshared Ada programs are initiated by loading a pre-initialized OS/32 task image whose sharable pure segment includes the standard Ada run time system. The start-up code of the task reads the blocks of code and data into its impure segments.

A limited number of host files are created and allocated when the KAPSE is installed, for the purpose of holding OS/32 task images with sharable segments. When sharing is warranted, the Data Base Manager task copies the pure and impure parts of the Ada program into the file in Task Establisher Task (TET) format, and then uses that file for task loading [N-2]. These files are re-used dynamically on a LRU basis.

3.2.9.7 Inter-Program Communication

(a) Specification.

```
procedure IPC_SEND(ID: in PROGRAM_ID; DATA: in BUFFER_PTR);
     -- Send the message to the designated program.
     -- The program will receive a pseudo interrupt
     --  when the message is received, and access
     --  to a copy of the data.  The data will
     --  be drained as soon as the communicaiton
     --  is successful.
     -- IPC_SEND automatically records the PROGRAM_ID
     --  of the sending program within the data
     --  received.

  ...

end KAPSE_HOST_INTERFACE;
```

(b) Internal Representation and Algorithms --VM/370. All communication between virtual machines is accomplished using the Virtual Machine Communication Facility. This provides an interrupt to the receiving VM when a message is ready. The data is copied using a fast memory to memory transfer [N-1].

(c) Internal Representation and Algorithms --OS/32. Communication between OS/32 tasks uses the task message facility. Pseudo interrupts are provided to the receiving task when a message is ready.

For large transfers, OS/32 provides the ability to send and receive open file handles. If the overhead of messages becomes unwieldy in a running MAPSE, it will be possible to switch to a method of data transfer involving writing to a scratch file from one task, and then reading the data back in the receiving task [N-2].

3.3 Adaptation and Rehosting

3.3.1 Installation parameters

The following parameters must be supplied as part of installing a KAPSE on a particular host:

1. The block size;
2. The number of block buffers in the buffer cache;
3. The maximum number of simultaneous users;
4. The maximum number of simultaneous programs.

3.3.2 Operation parameters

The following parameters may be adjusted on a running KAPSE to reflect a changing operational environment:

1. The maximum memory allocation per program;
2. The limit on number of simultaneous programs per user;
3. Host-dependent scheduling parameters;
4. The names and numbers of device objects (see 3.2.3.1).
5. Processing and disk budgets (see 3.2.7.3).

3.3.3 System Capacities

KAPSE performance will vary according to user load and host system speed and capacity. In addition to the above installation and operation parameters, the following parameters will have a significant impact on throughput and response time:

1. The current number of simultaneous programs;
2. The amount of database access;
3. The locality of database access;
4. The amount of inter-program communication;
5. The number of simultaneous interactive users.

3.3.4 Rehosting Requirements

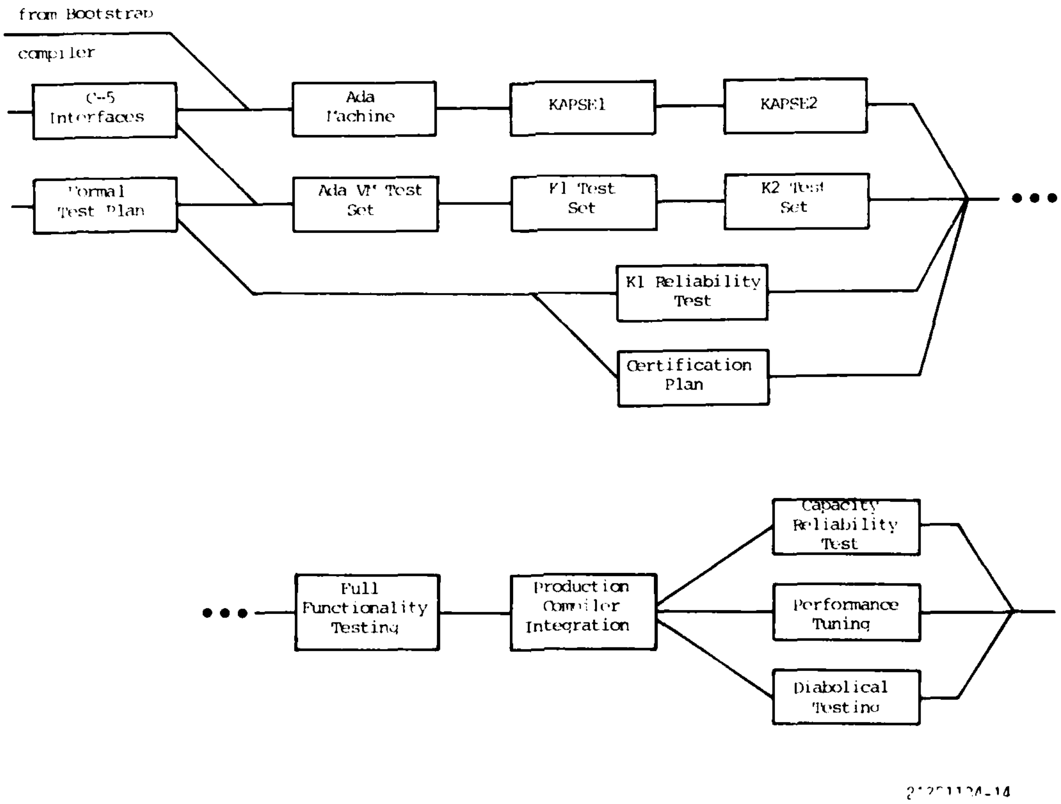
Rehosting the KAPSE will require retargeting the Ada compiler and re-implementing the KAPSE/Host interface. The KAPSE/Host interface has been kept as simple and low-level as possible to facilitate rehosting to a new host system or bare machine.

Any host must provide some kind of direct access disk or other on-line storage device. The host must also provide some kind of asynchronous pseudo interrupt to implement Ada real-time constructs and inter-program communication.

4. QUALITY ASSURANCE

Because the KAPSE serves as the guardian of the entire database, the testing and validation procedure must be very intensive. The general approach is to use automation and parallel efforts to achieve a high level of confidence in a short time. These activities are illustrated below:

KAPSE Sub-Project



4.1 Ada Machine Testing

The Ada machine consists of implementations of all routines called implicitly by Ada programs, and the specifications and bodies of all subprograms defined by the standard environment. It includes heap management, tasking, and all other Ada operations defined by the language as well as the KAPSE/VM370 interface routines (the first host). The Ada machine test set consists of the ACVS compiler validation set.

4.2 Production Input/Output Tests

The next test/production phase covers basic database functions below the user level. These include the following functions:

1. KAPSE/Host interfaces;
2. Disk volume identification/initialization;
3. Disk block allocation and reference-counting;
4. Disk block read, write, copy;
5. Block totals, join-counting, reserve/release;
6. Composite objects and distinguishing attributes -- creation and deletion of simple object components;
7. Non-distinguishing attributes;
8. A primitive history attribute, logging all KAPSE calls.

When the units listed above have been tested individually, the project begins to develop the production software on the system developed so far, rather than the bootstrap environment. Database integrity is the responsibility of a human software librarian, who does manual backups daily. "Self-use," or further development of the KAPSE on the KAPSE, is the primary form of integration testing at this point.

4.3 KAPSE Version 1 Test Case Generation

The scripts saved during this phase, especially those which failed or caused a system crash, will become the primary set of regression tests. The MAPSE project manager will run the regression and other tests and commit the entire KAPSE/MAPSE project to the use of "KAPSE-1" as a development system, after the following additional features have been developed:

1. Categories;
2. All remaining operations on components of composite objects;
3. Partitions;
4. Access rights and capacities;
5. Windows stored in the database;
6. Automatic backup and recovery;

The combined set of unit, integration, and regression tests developed by this point are a proposed AIE validation set (PAVS). They are used as an acceptance test for new releases of the KAPSE to the rest of the AIE project. A program will be developed to automatically run this test set once, or repeatedly, and check for correct execution of all tests.

4.4 K1 Reliability Test

The PAVS tests will be run cyclicly on the version 1 KAPSE for two weeks without crashing. It is estimated that four weeks of calendar time will be needed to debug version 1 to the point of surviving two weeks. This reliability testing overlaps additional development work in the areas of:

1. The remainder of the history mechanisms;
2. Complete configuration management primitives;
3. User budgets and accounting;
4. Login/Logout;
5. Management-oriented data maintenance;
6. Full terminal handling software.

4.5 Full Function Testing

Afte incorporating any changes indicated by the outcome of K1 reliability testing, and the list of new developments above, KAPSE version two and test set K2 are developed. Set K2 includes set K1, specific unit tests for the new features, scripts saved from all K1 crashes, and any other tests which will be required for government acceptance. Testing and debugging are continued until all K2 tests have been passed. Next the KAPSE is recompiled with the production compiler, and set K2 is repeated. KAPSE version three consists of version two as recompiled and re-debugged with respect to test set K2.

4.6 KAPSE Version 3 Testing

Version three testing will proceed as three parallel efforts: The first will be the capacity and reliability test, consisting of running the full K2 set continuously for two weeks with a database constantly growing in number of objects, users, categories, etc. At the same time, there will be diabolical testing, consisting of giving skilled programmers specific instruction and motivation to find ways to defeat access controls, corrupt the database, etc. And finally, as programmers make corrections and performance improvements, they will perform development testing.

4.7 Acceptance Testing

The acceptance test consists of the K2 set, the capacity and reliability test, the scripts generated during successful and unsuccessful diabolical tests, and throughput tests to measure performance against the level A requirements.

10. APPENDIX

10.1 Package INPUT OUTPUT in Ada

This is a preliminary effort to implement Package INPUT_OUTPUT in Ada, as a model for any machine language implementation:

```
with KAPSE_TYPES;
generic
     type ELEMENT_TYPE is limited private;
package INPUT_OUTPUT is
     type IN_FILE    is limited private;
     type OUT_FILE   is limited private;
     type INOUT_FILE is limited private;

     type FILE_INDEX is range KAPSE_TYPES.FILE_INDEX'RANGE;

     -- general operations for file manipulation

 ... -- As in Ada LRM

private
     -- declarations of the file private types
     type IN_FILE    is new FILE_HANDLE(ELEMENT_TYPE'SIZE);
     type OUT_FILE   is new FILE_HANDLE(ELEMENT_TYPE'SIZE);
     type INOUT_FILE is new FILE_HANDLE(ELEMENT_TYPE'SIZE);

end INPUT_OUTPUT;


package KAPSE_TYPES is
     FH_CLOSED: constant -1;  -- initial value for file handles.
     type FILE_HANDLE(SIZE_IN_BITS:INTEGER := 0) is
          record
               FH_INDEX: INTEGER := FH_CLOSED;
          end record;
     type FILE_INDEX is range 0..(2**31)-1;
     type FILE_NAME(LEN:0..256 := 0) is
          record
               NAME:STRING(1..LEN);
          end record;
     subtype ADDRESS_TYPE is LONG_INTEGER;
     type KAPSE_STATUS is
          (NO_ERROR, NAME_ERROR, USE_ERROR, STATUS_ERROR,
               DATA_ERROR, DEVICE_ERROR, END_ERROR);
     type FILE_MODE is (IN_MODE, INOUT_MODE, CR_INOUT_MODE,
               CR_OUT_MODE, OUT_MODE);
     type KAPSE_OPERATION is (CREATE_OPEN, GET_PUT_STAT, READ_WRITE,
                    etc...);
```


INTERMETRICS INCORPORATED • 733 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

```
     type KAPSE_ARG(OPERATION:KAPSE_OPERATION) is
          record
               STATUS: KAPSE_STATUS := NO_ERROR;
               case OPERATION is
                    when CREATE_OPEN =>
                     OPEN_FH: FILE_HANDLE;
                     OPEN_NAME: FILE_NAME;
                     OPEN_MODE: FILE_MODE;
                    when CLOSE =>
                     CLOSE_FH: FILE_HANDLE;
                    when GET_PUT_STAT =>
                     STAT_PUT_FLAG: BOOLEAN := FALSE;
                     STAT_FH: FILE_HANDLE;
                     STAT_SIZE: FILE_INDEX;
                     STAT_FIRST: FILE_INDEX;
                     STAT_LAST: FILE_INDEX;
                     STAT_NEXT_READ: FILE_INDEX;
                     STAT_NEXT_WRITE: FILE_INDEX;
                    when READ_WRITE =>
                     WRITE_FLAG: BOOLEAN := FALSE;
                     RW_FH: FILE_HANDLE;
                     RW_ADDRESS: ADDRESS_TYPE;
                     RW_SIZE: INTEGER;
               ...
               end case;
          end record;
end KAPSE_TYPES;


with KAPSE_TYPES,GENIO; use KAPSE_TYPES;
package body INPUT_OUTPUT is

     procedure RAISE_EXCEPTION(STATUS: KAPSE_STATUS) is
     begin
          case STATUS is
               when NAME_ERROR  => raise NAME_ERROR;
               when USE_ERROR  => raise USE_ERROR;
               when STATUS_ERROR => raise STATUS_ERROR;
               when DATA_ERROR  => raise DATA_ERROR;
               when DEVICE_ERROR => raise DEVICE_ERROR;
               when END_ERROR  => raise END_ERROR;

               when others  => null;

          end case;
     end RAISE_EXCEPTIONS;

     -- Typical implementations for file manipulation procedures.
     --  This is only a representative sample.

     procedure CREATE(FILE:in out INOUT_FILE; NAME:in STRING) is
          STATUS: KAPSE_STATUS;
     begin
          GENIO.CREATE_OPEN(FILE_HANDLE(FILE), NAME,
              CR_INOUT_MODE, STATUS);
          RAISE_EXCEPTION(STATUS);
     end CREATE;
```

```
procedure OPEN (FILE:in out INOUT_FILE; NAME:in STRING) is
     STATUS: KAPSE_STATUS;
begin
     GENIO.CREATE_OPEN(FILE_HANDLE(FILE), NAME,
          INOUT_MODE, STATUS);
     RAISE_EXCEPTION(STATUS);
end OPEN;

procedure CLOSE (FILE:in out OUT_FILE) is
     STATUS: KAPSE_STATUS;
begin
     GENIO.CLOSE(FILE_HANDLE(FILE), STATUS);
     RAISE_EXCEPTION(STATUS);
end CLOSE;

function  IS_OPEN(FILE:in INOUT_FILE)return BOOLEAN is
begin
     return GENIO.IS_OPEN(FILE_HANDLE(FILE));
end IS_OPEN;

-- Typical implementations for input and output operations.
--  This is only a representative sample.

procedure READ (FILE:in IN_FILE; ITEM:out ELEMENT_TYPE) is
     STATUS: KAPSE_STATUS;
     LOCAL_ITEM: ELEMENT_TYPE;
          -- Use local in case actual
          --  constrained.
begin
     GENIO.READ(FILE_HANDLE(FILE),
          LOCAL_ITEM'ADDRESS, STATUS);
     RAISE_EXCEPTION(STATUS);
     ITEM := LOCAL_ITEM;
          -- Might cause constraint exception
end READ;

function NEXT_READ (FILE:in IN_FILE) return FILE_INDEX is
     STATUS: KAPSE_STATUS;
     REC_NUM: FILE_INDEX;
begin
     GENIO.NEXT_READ(FILE_HANDLE(FILE), STATUS,
          KAPSE_TYPES.FILE_INDEX(REC_NUM));
     RAISE_EXCEPTION(STATUS);
     return REC_NUM;
end NEXT_READ;

procedure RESET_READ (FILE:in IN_FILE) is
     STATUS: KAPSE_STATUS;
begin
     GENIO.RESET_READ(FILE_HANDLE(FILE), STATUS);
     RAISE_EXCEPTION(STATUS);
end RESET_READ;
```

```
      procedure WRITE(FILE:in INOUT_FILE; ITEM:in ELEMENT_TYPE) is
           STATUS: KAPSE_STATUS;
      begin
           -- include ITEM'SIZE in WRITE in case
           --  variable length records.
           GENIO.WRITE(FILE_HANDLE(FILE), ITEM'ADDRESS,
                ITEM'SIZE, STATUS);
           RAISE_EXCEPTION(STATUS);
      end WRITE;


 end INPUT_OUTPUT;

 with KAPSE_TYPES; use KAPSE_TYPES;
 package body GENIO is

      -- Possible implementations for the generalized I/O
      --  operations.

      procedure CREATE_OPEN(FH:in out FILE_HANDLE;
          NAME:in STRING; MODE:in FILE_MODE;
          STATUS:out KAPSE_STATUS) is
           ARG: KAPSE_ARG(CREATE_OPEN);
      begin
           if FH.FH_INDEX /= FH_CLOSED then
                STATUS := STATUS_ERROR;
           else
                ARG.OPEN_NAME := (NAME'LENGTH,NAME);
                ARG.OPEN_MODE := MODE;
                ARG.OPEN_FH := FH;
                KAPSE_CALL(ARG);
                FH := ARG.OPEN_FH;
                STATUS := ARG.STATUS;
           endif;

      end CREATE_OPEN;

      procedure CLOSE(FH: in out FILE_HANDLE;
          STATUS: out KAPSE_STATUS) is
           ARG: KAPSE_ARG(CLOSE);
      begin
           if FH.FH_INDEX = FH_CLOSED then
                STATUS := STATUS_ERROR;
           else
                ARG.CLOSE_FH := FH;
                KAPSE_CALL(ARG);
                FH := ARG.CLOSE_FH;
                STATUS := ARG.STATUS;
           endif;
      end CLOSE;
```

```
procedure NEXT_READ(FH: in FILE_HANDLE;
    STATUS: out KAPSE_STATUS; REC_NUM: out FILE_INDEX) is
     ARG: KAPSE_ARG(GET_PUT_STAT);
begin
     ARG.STAT_FH := FH;
     KAPSE_CALL(ARG);
     STATUS := ARG.STATUS;
     REC_NUM := ARG.STAT_NEXT_READ;
end NEXT_READ;

procedure END_OF_FILE(FH: in FILE_HANDLE;
    STATUS: out KAPSE_STATUS; AT_END: out BOOLEAN) is
     ARG: KAPSE_ARG(GET_PUT_STAT);
begin
     ARG.STAT_FH := FH;
     KAPSE_CALL(ARG);
     STATUS := ARG.STATUS;
     AT_END := (ARG.STAT_NEXT_READ > ARG.STAT_LAST);
end END_OF_FILE;

procedure RESET_READ(FH: in FILE_HANDLE;
    STATUS: out KAPSE_STATUS) is
     ARG: KAPSE_ARG(GET_PUT_STAT);
begin
     ARG.STAT_FH := FH;
     KAPSE_CALL(ARG);
     STATUS := ARG.STATUS;
     if STATUS = NO_ERROR then
          ARG.STAT_NEXT_READ := ARG.STAT_FIRST;
          ARG.STAT_PUT_FLAG := TRUE;
          KAPSE_CALL(ARG);
          STATUS := ARG.STATUS;
     endif
end RESET_READ;

function IS_OPEN(FH: in FILE_HANDLE) return BOOLEAN is
begin
     return (FH.FILE_INDEX /= FH_CLOSED);
end IS_OPEN;

procedure READ(FH:FILE_HANDLE; ADDR:ADDRESS_TYPE;
    STATUS: out KAPSE_STATUS) is
     ARG: KAPSE_ARG(READ_WRITE);
begin
     ARG.RW_FH := FH;
     ARG.RW_ADDR := ADDR;
     ARG.RW_SIZE := FH.SIZE_IN_BITS;
     KAPSE_CALL(ARG);
     STATUS := ARG.STATUS;
end READ;
```

```
procedure WRITE(FH: FILE_HANDLE; ADDR: ADDRESS_TYPE;
    SIZE: INTEGER; STATUS: out KAPSE_STATUS) is
     ARG: KAPSE_ARG(READ_WRITE);
begin
     ARG.WRITE_FLAG := TRUE;
     ARG.RW_FH := FH;
     ARG.RW_ADDR := ADDR;
     ARG.RW_SIZE := SIZE;
     KAPSE_CALL(ARG);
     STATUS := ARG.STATUS;
end WRITE;

end GENIO;
```

10.2 Ada Tasking Model in Ada

This is an attempt to model the Ada Task Scheduler in Ada, to aid in any eventual machine language implementation. This design is not complete, and is provided here only to indicate the direction of the implementation.

This is to be read in conjunction with the Ada code for the task scheduling routines, found later in this appendix. For the various tasking constructs, the appropriate sets of calls to the task scheduler are listed below:

1) Simple DELAY statement:

```
Ada: DELAY simple_expression;

Code:     D := simple_expression;
     SET_DELAY(AMT=>D, WAIT_NOW=>TRUE);
```

2) Simple ACCEPT statement:

```
Ada: ACCEPT entry_id [(index_within_family)] [formal_part]
          [ DO sequence_of_statements END [entry_id]];

Code:     [ I := index_within_family; ]
     E := <entry_num_base_of entry_id> [ + I ];
     SIMPLE_ACCEPT(ENTRY_NUM=>E);
     S := RUNNING_TSK.CALLER_STACK;
     [ sequence_of_statements ]
          -- Use S as base for access to actual parameters
     END_RENDEZVOUS;
```

3) Simple ENTRY call:

```
Ada: task.entry_id [(index_within_family)]
                         [(actual_parameter_part)];

Code:     T := <pointer_to_tcb_of task>;
     [ I := index_within_family; ]
     E := <entry_num_base_of entry_id> [ + I ];
     [ <put_on_stack actual_parameter_part>; ]
     ENTRY_CALL(TSK=>T, ENTRY_NUM=>E, WAIT_IF_NOT_AVAIL=>TRUE);
     CASE RUNNING_TSK.CALL_STATUS OF
       WHEN EXCEPTION_RAISED =>
          RAISE RUNNING_TSK.EXCEPTION_NUMBER;
               -- (Not strict Ada)
       WHEN NORMAL_CALL =>
          NULL;
       WHEN OTHERS =>
          RAISE TASKING_ERROR;
     END CASE;
```


INTERMETRICS INCORPORATED • 733 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

4) Conditional ENTRY call:

```
Ada: SELECT simple_entry_call [ sequence_of_statements ]
        ELSE sequence_of_statements
        END SELECT;

Code:     T,I,E := as above for Simple ENTRY call;
     [ <put_on_stack actual_parameter_part>; ]
     ENTRY_CALL(TSK=>T, ENTRY_NUM=>E, WAIT_IF_NOT_AVAIL=>FALSE);
     CASE RUNNING_TSK.CALL_STATUS OF
       WHEN EXCEPTION_RAISED =>
          RAISE RUNNING_TSK.EXCEPTION_NUMBER;
              -- (Not strict Ada)
       WHEN NORMAL_CALL =>
          [ sequence_of_statements ]
       WHEN NOT_AVAIL =>
          sequence_of_statements
       WHEN OTHERS =>
          RAISE TASKING_ERROR;
     END CASE;
```

5) Timed ENTRY call:

```
Ada: SELECT simple_entry_call [ sequence_of_statements ]
        OR DELAY delay_amount; [ sequence_of_statements ]
        END SELECT;

Code:     T,I,E := as above for Simple ENTRY call;
     [ <put_on_stack actual_parameter_part>; ]
     D := delay_amount;
     SET_DELAY(DELAY_AMT=>D, WAIT_NOW=>FALSE);
     ENTRY_CALL(TSK=>T, ENTRY_NUM=>E, WAIT_IF_NOT_AVAIL=>TRUE);
     CASE RUNNING_TSK.CALL_STATUS OF
       WHEN EXCEPTION_RAISED =>
          RAISE RUNNING_TSK.EXCEPTION_NUMBER;
              -- (Not strict Ada)
       WHEN NORMAL_CALL =>
          [ sequence_of_statements ]
       WHEN DELAY_TIME_UP =>
             [ sequence_of_statements ]
       WHEN OTHERS =>
          RAISE TASKING_ERROR;
     END CASE;
```

6) SELECT statement:

```
Ada: SELECT
       [WHEN condition =>] select_alternative
    OR [WHEN condition =>] select_alternative  OR ...
       [ELSE sequence_of_statements]
     END SELECT;
   select_alternative ::=
     accept_statement [sequence_of_statements] |
     delay_statement [sequence_of_statements] |
     TERMINATE;
```

```
Code:     DECLARE
          NA: CONSTANT := <number_of_select_alternatives>;
          TYPE DELAY_INFO IS RECORD
               DELAY_AMT: DURATION;
               SLCT_INDEX: 0..NA := 0;
          END_RECORD;
          SHORTEST_DELAY: DELAY_INFO;
          WAIT_FLAG: BOOLEAN := <FALSE if ELSE part present>;
          INDEX: 1..NA+1;
          S: ADDRESS_TYPE;  -- Will hold caller's stack ptr.

     BEGIN
          <generate initial code for each alternative>
                -- See below for initial code sequences.
          IF SHORTEST_DELAY.SLCT_INDEX /= 0 THEN
               IF SHORTEST_DELAY.DELAY_AMT <= 0 THEN
               WAIT_FLAG := FALSE;
               ELSE
               SET_DELAY(DELAY_AMT=>
                   SHORTEST_DELAY.DELAY_AMT,
                   WAIT_NOW=>FALSE);
               END IF;
          END IF;
          SELECT_CALLER(WAIT_IF_NONE=>WAIT_FLAG);
          CASE RUNNING_TSK.CALL_STATUS OF
            WHEN NONE_READY|NONE_OPEN|DELAY_TIME_UP =>
               [ INDEX := NA + 1; ]  -- refers to ELSE part.
               [ INDEX := SHORTEST_DELAY.INDEX; ]
               -- Choose one of above, first if
               --  ELSE part is present, second
               --  if not.
            WHEN SELECT_SUCCESSFUL =>
               INDEX := RUNNING_TSK.SLCT_INDEX;
            WHEN OTHERS =>
               RAISE SELECT_ERROR;  -- None open.
          END CASE;
          S := RUNNING_TSK.CALLER_STACK;
               -- S is pointer to caller's stack.
          CASE INDEX OF
            WHEN 1 =>
               [ alternative_1_rendezvous_code
               -- Use S to access actual arguments.
               END_RENDEZVOUS; ]
               -- Above is not present
               --   if Delay alternative.
               [ alternative_1_sequence_of_statements ]
            WHEN 2 =>
               [ alternative_2_rendezvous_code
               END_RENDEZVOUS; ]  -- Not present if Delay.
               [ alternative_2_sequence_of_statements ]
           ...
            WHEN NA+1 =>
               [ else_part_sequence_of_statements ]
          END CASE;
     END;
```


INTERMETRICS INCORPORATED • 733 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

Initial code for alternatives:

ACCEPT alternatives:

```
[ IF condition THEN ] -- Only present if WHEN present.
    [ I := index_within_family; ]
    E := <entry_num_base_of entry_id> [ + I ];
    IX := index_of_alternative;
    SET_OPEN(ENTRY_NUM=>E, SLCT_INDEX=>IX);
[ END IF; ]
```

DELAY alternatives:

```
[ IF condition THEN ] -- Only present if WHEN present.
    D := delay_amount;
    IX := index_of_alternative;
    IF SHORTEST_DELAY.SLCT_INDEX = 0 OR ELSE
        D < SHORTEST_DELAY.DELAY_AMT THEN
         SHORTEST_DELAY := (DELAY_AMT=>D,SLCT_INDEX=>IX);
    END IF;
[ END IF; ]
```

TERMINATE alternatives:

```
[ IF condition THEN ] -- Only present if WHEN present.
    READY_TO_TERMINATE;
[ END IF; ]
```

6) Task Activation:

[Not completed yet]

INTERMETRICS INCORPORATED • 733 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

This is the actual Ada code for a possible implementation of the Ada task scheduler. This is very preliminary and as yet incomplete.

```
Package body TASK_SCHEDULER is

     type TCB(NUM_ENTRIES: INTEGER);-- incomplete for now

     type TCB_PTR is access TCB;

     RUNNING_TSK: TCB_PTR := null;

     type BIT_VEC is array(NATURAL range <>) of BOOLEAN;

     type TASK_STATE is (INACTIVE,RUNNABLE,RUNNING,WAIT_FOR_ACCEPT,
          WAIT_FOR_CALLER,CALLER_IN_RENDEZVOUS,TERMINATED);
               -- Delay is encoded separately

     type TASK_CALL_STATUS is (NORMAL_CALL,EXCEPTION_RAISED,
         DELAY_TIME_UP, NOT_AVAIL,NONE_OPEN,NONE_READY,ABNORMAL);

     type LINK is record
          NEXT: TCB_PTR := null;
          PREV: TCB_PTR := null;
     end record;

     type QUEUE_NAMES is (CURRENT,SIBLING,DELAYQ);
     type HEADER(NAME: QUEUE_NAMES) is record
          COUNT: INTEGER := 0;
          FIRST: TCB_PTR := null;
          LAST: TCB_PTR  := null;
     end record;

     type PER_ENTRY_INFO is record
          SLCT_INDEX: INTEGER;
          QUEUE: HEADER(CURRENT);
     end record;

     type PROCESS_STATE is record
          STARTUP_ADDR: ADDRESS_TYPE;
          STACK_POINTER: ADDRESS_TYPE;
     end record;
```


INTERMETRICS INCORPORATED • 733 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

```
--   Task Control Block
type TCB(NUM_ENTRIES: INTEGER) is
     record
          PRIO: PRIORITY;
          STATE: TASK_STATE := INACTIVE;
          SAVED_PROCESS_STATE: PROCESS_STATE;
          DELAY_SET: BOOLEAN := FALSE;
          DELAY_DIFF: INTEGER;
              -- Decremented on clock tick.
          CALLING: TCB_PTR;
              -- Null if not WAIT_FOR_ACCEPT
          CALL_ENTRY_NUM: INTEGER;
          CALL_STATUS: TASK_CALL_STATUS;
          CALLER_STACK: ADDRESS_TYPE;
              -- Will hold caller's S.P.
          EXCEPTION_NUMBER: INTEGER;
          RENDEZVOUS_QUEUE: HEADER(CURRENT);
              -- List of callers
              --  now in rendezvous.
          LINKS: array(QUEUE_NAMES) of LINK;
          OPEN_ENTRIES: BIT_VEC(1..NUM_ENTRIES) :=
              (1..NUM_ENTRIES => FALSE);
               -- TRUE-->ACCEPT OPEN
          ENTRY_QUEUES: array(1..NUM_ENTRIES) of
              PER_ENTRY_INFO;
     end record;

RUN_QUEUES: array(PRIORITY) of HEADER(CURRENT);
     -- Array of run queues, ordered by priority.

DELAY_QUEUE: HEADER(DELAYQ);-- List of tasks with delay set.


procedure APPEND (ELEM: TCB_PTR; QUEUE: in out HEADER) is
     QX: constant QUEUE_NAMES := QUEUE.NAME;
     ELEM_LINK: LINK renames ELEM.LINKS(QX);

begin          -- Append element to end of doubly-linked list.

     ELEM_LINK.NEXT := null;
     if QUEUE.LAST = null then
          QUEUE.FIRST := ELEM;
          QUEUE.COUNT := 1;
          ELEM_LINK.PREV := null;
     else
          QUEUE.LAST.LINKS(QX).NEXT := ELEM;
          QUEUE.COUNT := QUEUE.COUNT + 1;
          ELEM_LINK.PREV := QUEUE.LAST;
     end if;
     QUEUE.LAST := ELEM;
end APPEND;
```

```
procedure INSERT (ELEM: TCB_PTR; QUEUE: in out HEADER;
          BEFORE: TCB_PTR) is
     QX: constant QUEUE_NAMES := QUEUE.NAME;
     ELEM_LINK: LINK renames ELEM.LINKS(QX);

begin          -- Insert element in middle of doubly-linked list.

     if BEFORE = null then
          APPEND(ELEM, QUEUE);
     else
          ELEM_LINK.PREV := BEFORE.LINKS(QX).PREV;
          ELEM_LINK.NEXT := BEFORE;
          BEFORE.LINKS(QX).PREV := ELEM;
          if ELEM_LINK.PREV /= null then
               ELEM_LINK.PREV.LINKS(QX).NEXT := ELEM;
          else
               QUEUE.FIRST := ELEM;
          end if;
          QUEUE.COUNT := QUEUE.COUNT + 1;
     end if;
end INSERT;

procedure PREPEND (ELEM: TCB_PTR; QUEUE: in out HEADER) is

begin          -- Insert element at beginning of queue.

     INSERT(ELEM, QUEUE, QUEUE.FIRST);
end PREPEND;

procedure REMOVE (ELEM: TCB_PTR; QUEUE: in out HEADER) is
     QX: constant QUEUE_NAMES := QUEUE.NAME;
     ELEM_LINK: LINK renames ELEM.LINKS(QX);

begin          -- Remove element from doubly-linked list.

     if ELEM_LINK.PREV = null then
          QUEUE.FIRST := ELEM_LINK.NEXT;
     else
          ELEM_LINK.PREV.LINKS(QX).NEXT := ELEM_LINK.NEXT;
     end if;
     if ELEM_LINK.NEXT = null then
          QUEUE.LAST := ELEM_LINK.PREV;
     else
          ELEM_LINK.NEXT.LINKS(QX).PREV := ELEM_LINK.PREV;
     end if;
     ELEM_LINK := (null,null);
     QUEUE.COUNT := QUEUE.COUNT - 1;
end REMOVE;
```

```
procedure FIRST_ELEM (ELEM: out TCB_PTR; QUEUE: in out HEADER) is

begin           -- Remove first element of doubly-linked list.

     if QUEUE.FIRST = null then
          ELEM := null;
     else
          ELEM := QUEUE.FIRST;
          REMOVE(ELEM, QUEUE);
     end if;
end NEXT_ELEM;

function NEXT_TO_RUN return TCB_PTR is
     TSK: TCB_PTR;
begin           -- Return TCB_PTR for highest priority runnable task.

     for PRIO in reverse PRIORITY loop
          if RUN_QUEUES(PRIO).COUNT > 0 then
               FIRST_ELEM(TSK, RUN_QUEUES(PRIO));
               TSK.STATE := RUNNING;
               return TSK;
          end if;
     end loop;
     return null;
end NEXT_TO_RUN;

procedure SET_DELAY (DELAY_AMOUNT: DURATION; WAIT_NOW: BOOLEAN) is
     TICKS: INTEGER;
     TP: TCB_PTR;

begin           -- Add task to delay queue at appropriate point.

     TICKS := DELAY_AMOUNT*TICKS_PER_SECOND + 1;
          -- "+1" necessary to guarantee "at least" proper delay.
     TP := DELAY_QUEUE.FIRST;
     while TP /= null and then TICKS >= TP.DELAY_DIFF loop
          TICKS := TICKS - TP.DELAY_DIFF;
          TP := TP.LINKS(DELAYQ).NEXT;
     end loop;
     RUNNING_TSK.DELAY_DIFF := TICKS;
     if TP /= null then
          TP.DELAY_DIFF := TP.DELAY_DIFF - TICKS;
          INSERT(RUNNING_TSK, DELAY_QUEUE, TP);
     else
          APPEND(RUNNING_TSK, DELAY_QUEUE);
     end if;
     RUNNING_TSK.DELAY_SET := TRUE;
     if WAIT_NOW then
          GIVE_UP_PROCESSOR;
     end if;
end SET_DELAY;
```

```
procedure CLEAR_DELAY(TSK: TCB_PTR) is
     TP: TCB_PTR;

begin           -- Remove from delay queue, if necessary.

     if TSK.DELAY_SET then
          TP := TSK.LINKS(DELAYQ).NEXT;
          if TP /= null then
               -- Adjust DIFF of following task on delay queue.
               TP.DELAY_DIFF := TP.DELAY_DIFF + TSK.DELAY_DIFF;
          end if;
          REMOVE(TSK, DELAY_QUEUE);
          TSK.DELAY_SET := FALSE;
     end if;
end CLEAR_DELAY;
procedure TICK is
     TP: TCB_PTR;

begin           -- Advance delay queue entries one tick.

     TP := DELAY_QUEUE.FIRST;
     if TP /= null then
          TP.DELAY_DIFF := TP.DELAY_DIFF-1;
          while TP /= null and then TP.DELAY_DIFF <= 0 loop
               TP.CALL_STATUS := DELAY_TIME_UP;
               SET_RUNNABLE(TP);
               TP := DELAY_QUEUE.FIRST;
          end loop;
     end if;
end TICK;

procedure REMOVE_FROM_QUEUES(TSK: TCB_PTR) is

begin           -- Remove from queues as necessary.

     CLEAR_DELAY(TSK);
     if TSK.STATE = WAIT_FOR_ACCEPT then
          REMOVE(TSK,
               TSK.CALLING.ENTRY_QUEUES(
                TSK.CALL_ENTRY_NUM).QUEUE);
     else
          TSK.OPEN_ENTRIES := (TSK.OPEN_ENTRIES'RANGE => FALSE);
     end if;
end REMOVE_FROM_QUEUES;
```

```
procedure SET_RUNNABLE(TSK: TCB_PTR) is

begin           -- Set task runnable, remove from queues as necessary.

     if TSK.STATE /= RUNNING then
          REMOVE_FROM_QUEUES(TSK);
     end if;
     if TSK.STATE /= TERMINATED and
         TSK.STATE /= RUNNABLE then
          TSK.STATE := RUNNABLE;
          -- Put on end of appropriate RUN queue.
          -- Changing the placement on the RUN queue
          --  could be used to adjust the within-priority
          --  scheduling algorithm.
          APPEND(TSK, RUN_QUEUES(TSK.PRIO));
     end if;
end SET_RUNNABLE;

procedure SET_OPEN(ENTRY_NUM: INTEGER; SLCT_INDEX: INTEGER) is

begin           -- Indicate Entry is open, save startup program ctr.

     RUNNING_TSK.ENTRY_QUEUES(ENTRY_NUM).SLCT_INDEX := SLCT_INDEX;
     RUNNING_TSK.OPEN_ENTRIES(ENTRY_NUM) := TRUE;
end SET_OPEN;

procedure ENTRY_CALL(TSK: TCB_PTR; ENTRY_NUM: INTEGER;
    WAIT_IF_NOT_AVAIL: BOOLEAN);
     ENT: PER_ENTRY_INFO renames TSK.ENTRY_QUEUES(ENTRY_NUM);

begin           -- Call particular entry.

     RUNNING_TSK.CALL_STATUS := NORMAL_CALL;
     if TSK.STATE = WAIT_FOR_CALLER and then
         TSK.OPEN_ENTRIES(ENTRY_NUM) then
          START_RENDEZVOUS(RUNNING_TSK,TSK,ENTRY_NUM);
          SET_RUNNABLE(TSK);
          GIVE_UP_PROCESSOR;
     elsif WAIT_IF_NOT_AVAIL then
          APPEND(RUNNING_TSK, ENT.QUEUE);
          RUNNING_TSK.STATE := WAIT_FOR_ACCEPT;
          GIVE_UP_PROCESSOR;
     else
          RUNNING_TSK.CALL_STATUS := NOT_AVAIL;
     end if;
end ENTRY_CALL;
```

```
procedure SELECT_CALLER(WAIT_IF_NONE:BOOLEAN) is

begin           -- Select caller, suspend if none unless COND true.

     if RUNNING_TSK.OPEN_ENTRIES =
          (RUNNING_TSK.OPEN_ENTRIES'RANGE => FALSE) then
          if WAIT_IF_NONE then
               raise SELECT_ERROR;
          else
               RUNNING_TSK.CALL_STATUS := NONE_OPEN;
               return;
          end if;
     end if;
     for I in RUNNING_TSK.OPEN_ENTRIES'RANGE loop
         declare
          ENT: ENTRY_INFO renames RUNNING_TSK.ENTRY_QUEUES(I);
         begin
          if RUNNING_TSK.OPEN_ENTRIES(I) and then
              ENT.QUEUE.COUNT > 0 then
               TP := ENT.QUEUE.FIRST;
               START_RENDEZVOUS(TP,RUNNING_TSK,I);
               RUNNING_TSK.CALL_STATUS := NORMAL_CALL;
               return;
          end if;
         end;
     end loop;
     if WAIT_IF_NONE then
          RUNNING_TSK.STATE := WAIT_FOR_CALLER;
          GIVE_UP_PROCESSOR;
     else
          RUNNING_TSK.CALL_STATUS := NONE_READY;
     end if;
end SELECT_CALLER;

procedure SIMPLE_ACCEPT(ENTRY_NUM:INTEGER) is
     ENT: ENTRY_INFO renames RUNNING_TSK.ENTRY_QUEUES(ENTRY_NUM);
     TP: TCB_PTR;

begin           -- Pick up next caller of this entry (FIFO).

     SET_OPEN(ENTRY_NUM);
     if ENT.QUEUE.COUNT > 0 then
          TP := ENT.QUEUE.FIRST;
          START_RENDEZVOUS(TP, RUNNING_TSK, ENTRY_NUM);
     else
          RUNNING_TSK.STATE := WAIT_FOR_CALLER;
          GIVE_UP_PROCESSOR;
     end if;
end SIMPLE_ACCEPT;
```

```
procedure START_RENDEZVOUS(CALLER:TCB_PTR; CALLED:TCB_PTR;
    ENTRY_NUM:INTEGER) is

begin          -- Start rendezvous between CALLER and CALLED task.

     REMOVE_FROM_QUEUES(CALLER);
     REMOVE_FROM_QUEUES(CALLED);
     PREPEND(CALLER, CALLED.RENDEZVOUS_QUEUE);  -- LIFO queue here.
     CALLER.STATE := CALLER_IN_RENDEZVOUS;
     CALLER.SAVED_PRIO := CALLED.PRIO;
     if CALLER.PRIO > CALLED.PRIO then
          -- Adjust CALLED task prio to MAX of the two.
          CALLED.PRIO := CALLER.PRIO;
     end if;
     CALLED.SLCT_INDEX := CALLED.ENTRY_QUEUES(ENTRY_NUM).SLCT_INDEX;
     CALLED.CALLER_STACK := CALLER.SAVED_PROCESS_STATE.STACK_POINTER;
end START_RENDEZVOUS;

procedure END_RENDEZVOUS is
     CALLER: TCB_PTR;

begin          -- Finish up rendezvous, use RENDEZVOUS queue to locate
          --  CALLER.

     FIRST_ELEM(CALLER, RUNNING_TSK.RENDEZVOUS_QUEUE);
     RUNNING_TSK.PRIO := CALLER.SAVED_PRIO;
     SET_RUNNABLE(CALLER);
     if CALLER.PRIO > RUNNING_TSK.PRIO then
          GIVE_UP_PROCESSOR;
     end if;
end END_RENDEZVOUS;
```

```
procedure GIVE_UP_PROCESSOR is

begin          -- Give up processor to allow others to run.
          -- Interrupts may force a call on GIVE_UP_PROCESSOR.

     SAVE_PROCESS_STATE(RUNNING_TSK.SAVED_PROCESS_STATE);
     if RUNNING_TSK.STATE = RUNNING then
          -- Still indicated as running,
          --   put back on RUNNABLE queue (this effects
          --   round-robin scheduling, with tiny time-slices).
          SETRUNNABLE(RUNNING_TSK);
     else
          -- Purposefully giving up processor,
          --  initialize CALL_STATUS.
          RUNNING_TSK.CALL_STATUS := NORMAL_CALL;
     end if;
     loop
          while TICKS > 0 loop  -- Incremented on clock interrupt.
               TICK;  -- May set a DELAYed task RUNNABLE.
               TICKS := TICKS - 1;
          end loop;
          RUNNING_TSK := NEXT_TO_RUN();
         exit when RUNNING_TSK /= null;
     end loop;
     RESTORE_PROCESS_STATE(RUNNING_TSK.SAVED_PROCESS_STATE);
end GIVE_UP_PROCESSOR;


end TASK_SCHEDULER;
```

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.

DATE
FILMED

–8